MASTER IN COMPUTER SCIENCE

An architecture and an Abstract Data Type for an Inductive Schema-Guided Logic Program Synthesizer

Bauvir, Christophe

Award date:
1996

Awarding institution:
University of Namur

An Architecture and an Abstract Data Type for an Inductive Schema-Guided Logic Program Synthesizer*

Christophe Bauvir

August 30, 1996

* This paper is the result of my work at Bilkent University (Ankara) under the supervision of Professor Pierre Flener and at the Facultés Universitaires Notre-Dame de la Paix (Namur) under the supervision of Professor Naji Habra. I would like to thank both of them for the continuous help they gave me and the interest they show for my work.

Résumé

Un *synthétiseur inductif de programmes logiques guidé par schéma* est un logiciel permettant d'automatiser le développement de programmes logiques à partir de spécifications. Pour synthétiser un programme logique, le logiciel est guidé par un *schéma*, qui englobe de la connaissance en conception d'algorithmes, et utilise principalement des règles d'inférence inductive pour extraire de l'information des spécifications. Premièrement, nous introduisons le domaine de recherche appelé la *synthèse de programmes* et donnons une vue d'ensemble de l'outil utilisé pour implémenter les différentes parties du synthétiseur. Deuxièmement, nous expliquons la notion de schéma et décrivons un type de donnée abstrait permettant de représenter des schémas ainsi que des services créant, manipulant et *instantiant* des schémas. Finalement, nous proposons une architecture modulaire pour un synthétiseur inductif de programmes logiques guidé par schéma appelé **DIALOGS**.

Abstract

An *inductive schema-guided logic program synthesizer* is a software allowing to automate the development of logic programs from specifications. To synthesize a logic program, the software is guided thanks to a *schema*, which embodies algorithm design knowledge, and mainly uses inductive inference rules to extract information from specifications. First, we introduce the *program synthesis* field of research and give an overview of the tool used to implement the different parts of the synthesizer. Second, we explain the notion of schema and describe an abstract data type representing schemas together with services to construct, manipulate and *instantiate* schemas. Finally, we propose a modular architecture for an inductive schema-guided logic program synthesizer called **DIALOGS**.

Contents

Chapter 1

Introduction to Program Synthesis

The *programming task* is viewed as a sequence of activities starting from non-formal intentions and finally producing a program. This programming task is presented via the following *software life-cycle*:

- *elaboration* of a specification from non-formal intentions;
- *development* of a program that is correct with respect to the specification.

The development phase is not always achieved and often produces a program that *seems* correct with respect to the specification. A third phase is needed:

- *validation* of the program so that it is correct with respect to the specification.

This validation phase can be done using two approaches. A first approach is called *retrospective program verification* where one checks the correctness of a given program with respect to its specification. This approach is extremely hard and difficult to put in practice with huge programs. The other approach is *program testing* and consists of comparing the current results given by a program from specific inputs with the expected results obtained from the same inputs when examining the specification of the program. This second approach cannot guarantee that the program will give the correct results with all possible inputs.
Moreover, specifications must evolve over time as intentions change. A fourth phase is, then, added to our software life-cycle:

- *maintenance* of the specification and *maintenance* of the program so that it is correct with respect to the new specification.

Program synthesis, also called *automatic programming*, is a field of research concerned with the automation of the *development* phase of the software life-cycle. The main idea of program synthesis is to build a software, called a *synthesizer*, developping correct programs from specifications. The benefits of the program synthesis approach is a disappearance of the validation phase from the software life-cycle. The development phase being less costly thanks to the use of synthesizers, a programmer will have the opportunity to mainly focus on the "elaboration of a specification" phase.
A wide variety of synthesizers have been developped. The synthesizers can be classified according to the predominent kind of reasoning used to extract information from specifications.
Here is a possible classification:

- inductive synthesizers;
- deductive synthesizers;
- abductive synthesizers;
- analogical synthesizers.

Remark: For a general description of inductive and deductive synthesizers, see [4].
A synthesizer, like a programmer, needs knowledge together with a specification to achieve its task.
This knowledge is mainly composed of:

- *algorithm design knowledge*;
- *domain knowledge* which is knowledge about the application domain of the future synthesized program;
- *meta-knowledge* which is knowledge about how and when to use the previous two knowledges;
- *efficiency knowledge* which is knowledge guiding the synthesizer to produce an efficient program.

The algorithm design knowledge can be codified using schemas. A *schema* is a template algorithm representing a whole family of particular algorithms designed according to a specific synthesis methodology like Divide-and-Conquer, Generate-and-Test, ...

The work presented in this paper consists of two parts. First, we define the notion of schema and describe an abstract data type allowing its representation. This abstract

data type is constructed thanks to *schema objects* which are detailed in section 3. Section 4 lists the services available to manipulate schemas and section 5 presents a tool to instantiate schemas: the *schema substitution.* Finally, we propose a modular architecture for an inductive schema-guided logic program synthesizer, called **DIALOGS**, composed of five blocks: the *specification* block, the *schemas creation* block, the *methods* block, the *strategies* block and the *synthesizer* block. An overview of Gödel, which was chosen as the tool to implement the parts of the synthesizer, follows this section.

1.1 An Overview of Gödel

Gödel is a relational programming language offering a type system, a module system, control and meta-programming facilities. Gödel is the tool chosen to implement the synthesizer.

Gödel is a typed language based on many-sorted first-order logic. Each constant, function, proposition and predicate has to be specified with a type declaration. By contrast, variables have their types inferred from their context. Gödel allows the declaration of new types and the declaration of parametric types.

Gödel facilitates the development of huge programs thanks to the structurate view offered by the concept of *module.* A module hides implementation details and avoids problems like name clashes. A module is generally composed of two parts:

- the *export* part gathering the declarations of symbols (types, constants, functions, propositions and predicates) available for internal use and for use in other modules which import them;
- the *local* part gathering the declarations of symbols, available for internal use only, and the definition of propositions and predicates declared in this module.

Gödel has a flexible computation rule. It can be guided by means of DELAY declarations which cause certain calls to be delayed until they are sufficiently instantiated. Gödel has also a pruning operator.

Gödel offers modules which provide meta-programming facilities via the ground representation. The module **Syntax** provides a number of abstract data types for representing object-level expressions and predicates for manipulating terms of these types. The module **Programs** provides the abstract data type PROGRAM for representing Gödel programs and predicates for manipulating terms of this type.

Remark: Gödel has adopted the opposite notation of Prolog to distinguish predicates and constants from variables. In Gödel, variable symbols start with a lowercase letter. Predicate and constant symbols start with an uppercase letter.

Remark: For a detailed description of the language Gödel, see [6], and for the ground representation used in Gödel, see [1] and [5].

Part I

An Abstract Data Type for Logic Algorithm Schemas

First, we define the notion of schema and explain its purpose in a synthesis process. Second, we present the objects needed to construct schemas. Third, we list different services available to manipulate schemas. Fourth, we describe an important tool for instantiating schemas: the *schema substitution*. Finally, we conclude and state improvements for a future implementation of this abstract data type.

Chapter 2

Introduction

Algorithms can be classified according to their synthesis methodologies, such as Divide-and-Conquer, Generate-and-Test, Top-Down Decomposition, Global Search and so on. Informally, an *algorithm schema* is a template algorithm with a fixed control and data flow, but without specific indications about the actual parameters or the actual computations. An algorithm schema, thus, abstracts a whole family of particular algorithms that can be obtained by instantiating its placeholders to particular parameters or computations, using the specification, the algorithm synthesized so far and the constraints of the schema.

First, we define the notion of logic algorithm schema. Then, we give an example of logic algorithm schema representing the Divide-and-Conquer synthesis methodology.
Remark: This section is heavily inspired from the work of Pierre Flener, especially [3] and [4].

2.1 Logic Algorithm Schemas

Before defining the logic algorithm schema concept, we need the notion of logic algorithm.
Definition: A *logic algorithm* defining a predicate p/n is a closed well-formed formula of the form:

$$\forall X_1 \ldots \forall X_n \; p(X_1, \ldots, X_n) \Leftrightarrow \mathcal{B}[X_1, \ldots, X_n]$$

where the X_i are distinct variables ($i = 1..n$) and $\mathcal{B}$ is a well-formed formula in prenex disjunctive normal form, whose only free variables are $X_1, \ldots, X_n$. The atom $p(X_1, \ldots, X_n)$ is called the *head* and $\mathcal{B}[X_1, \ldots, X_n]$ is called the *body* of the logic algorithm.
Example: The member problem can be specified like this: $member(E, L)$ iff E is an element of L, where E is a term and L is a list. A possible logic algorithm for $member/2$ is as follows:

$$\forall E \; \forall L \; member(E, L) \Leftrightarrow \exists H \; \exists T$$

$$\begin{array}{lllll} & L = [] & \& & false & \\ \vee & L = [H|T] & \& & E = H & \\ \vee & L = [H|T] & \& & E \neq H & \& \quad member(E, T) \end{array}$$

Remark: In the following, we drop the universal quantifications in front of the heads, as well as any existential quantifications at the beginning of bodies of logic algorithms. For a detailed description of logic algorithms, see [2].

Informally, a *logic algorithm schema* is a second-order logic algorithm and its placeholders are first/second-order variables. A particular logic algorithm, called an *instance* of the schema, is then obtained by instantiating the variables of the schema.

Example: Here is a logic algorithm schema for the Generate-and-Test methodology:

$$R(X, Y) \Leftrightarrow Generate(X, Y) \ \& \ Test(Y)$$

where X and Y are the first-order variables of the schema and R, $Generate$ and $Test$ are the second-order variables of the schema (also called *predicate variables*).
The logic algorithm

$$sort(L, S) \Leftrightarrow permutation(L, S) \ \& \ ordered(S)$$

is an instance of this Generate-and-Test schema, namely via the second-order substitution $\{R/sort, X/L, Y/S, Generate/permutation, Test/ordered\}$.
This logic algorithm schema covers a set of logic algorithms restricted to binary predicates. To have the possibility to instantiate more complex logic algorithms, we must introduce *form variables* in our schemas together with first/second-order variables. The form variables allow predicate variables of any arity, conjunctions and disjunctions of any length to appear in a schema.

Definition: A *logic algorithm schema* defining a second-order predicate R is a well-formed formula of the form:

$$\forall X_1 \ldots \forall X_n \ R(X_1, \ldots, X_n) \Leftrightarrow \mathcal{B}[X_1, \ldots, X_n]$$

and a set $\mathcal{S}$ of constraints relating R and the variables of $\mathcal{B}$, where n is a form variable or a constant, the X_i are distinct variables ($i = 1..n$), R is a predicate variable and $\mathcal{B}$ is a well-formed formula in prenex disjunctive normal form, whose free first-order variables are $X_1, \ldots, X_n$. All predicate variables are free. The atom $R(X_1, \ldots, X_n)$ is called the *head* and $\mathcal{B}[X_1, \ldots, X_n]$ is called the *body* of the logic algorithm schema.

A logic algorithm schema without predicate variables and form variables is a logic algorithm.

Definition: An *instance* of a logic algorithm schema is a logic algorithm obtained by the following sequence of operations:

1. permutation of the parameters, conjuncts, disjuncts and quantifications of the schema;

2. application of a schema substitution to the resulting schema, such that all predicate variables and form variables are instantiated to first-order objects, and such that the constraints are satisfied;

3. application of unfold transformations.

This process is called *instantiation* of a schema.

The purpose of a logic algorithm schema is to guide synthesis according to a specific synthesis methodology. A logic algorithm schema embodies high-level programming knowledge under the form of the schema itself and the set of constraints attached to it.

2.2 An Example of Logic Algorithm Schema

We focus on the Divide-and-Conquer methodology to illustrate the notion of schema. The Divide-and-Conquer methodology solves a problem by the following steps:

1. *divide* a problem into similar subproblems, unless it can be trivially solved;

2. *conquer* the subproblems by solving them recursively;

3. *combine* the solutions to the subproblems into a solution to the initial problem.

A Divide-and-Conquer algorithm for a binary predicate r over parameters X and Y works as follows. Let X be the induction parameter. If X is minimal, then Y is (usually) easily found by directly solving the problem. Otherwise, that is if X is non-minimal, decompose X into a vector of variables **HX** of heads of X and into a vector of variables **TX** of tails of X, the tails being of the same type as X, as well as smaller than X according to some well-founded relation. The tails **TX** recursively yield tails **TY** of Y. The heads **HX** are processed into a vector of variables **HY** of heads of Y. Finally, Y is composed from its heads **HY** and tails **TY**.
We can quantify the vectors of variables as follows. There are hx heads of X, hy heads of Y and t tails of X, hence t tails of Y. We can note that hx, hy and t are form variables not constants (as introduced previously).

A logic algorithm schema obtained from the Divide-and-Conquer algorithm discussed above is as follows:

$$R(X, Y) \Leftrightarrow$$

$$
\begin{array}{rlll}
 & Minimal(X) & \& & Solve(X, Y) \\
\vee & NonMinimal(X) & \& & Decompose(X, \mathbf{HX}, \mathbf{TX}) \\
 & & \& & \mathbf{R}(\mathbf{TX}, \mathbf{TY}) \\
 & & \& & Process(\mathbf{HX}, \mathbf{HY}) \\
 & & \& & Compose(\mathbf{HY}, \mathbf{TY}, Y)
\end{array}
$$

where **R(TX,TY)** stands for $\bigwedge_{j=1}^{t} R(TX_j, TY_j)$ (j is called an *index*).
This logic algorithm schema is constructed with specific objects like vectors of variables representing a finite sequence of variables without a fixed length (e.g. **HX**), conjunctions (e.g. $\bigwedge_{j=1}^{t} R(TX_j, TY_j)$), ... These objects are explained in detail in the next section.

A set of constraints $\mathcal{S}$ relating R and the variables of the schema is attached to this Divide-and-Conquer schema. This set contains the following constraints:

- the instance of X, that is the selected induction parameter, must be of an inductive type. This means that $type(X, T)$ & $T \in \{integer, list, tree, \ldots\}$ must be valid. The decomposition of X into tails that are each smaller than X according to some well-founded relation would otherwise be impossible and the Divide-and-Conquer methodology not applicable.

- the minimal and non-minimal forms must be mutually exclusive over the domain of the induction parameter. This means that $Minimal(X) \Leftrightarrow \neg NonMinimal(X)$ must be valid.

- the decomposition of X must yield tails TX_i that are each smaller than X according to some well-founded relation $<$. This means that

$$Decompose(X, \mathbf{HX}, TX_1, \ldots, TX_t)$$

$$\Rightarrow \exists < \ \forall i \in \{1, \ldots, t\}\ TX_i < X$$

  must be valid. It ensures termination of the algorithm in the all-ground mode.

Remark: The Divide-and-Conquer schema presented here is not sufficient to cover a wide variety of logic algorithms but its purpose is to illustrate the concept defined in this section. Other examples of logic algorithm schemas can be found in [2], [3] and [4]. A functional algorithm schema for the Divide-and-Conquer methodology is proposed in [10].

- the schema objects have to allow the construction of complex schemas;
- there has to exist services to manipulate schema objects;
- the schema objects have to be easy to understand and to use.

We have defined schema objects as abstract data types, that is to say, we have services to create and manipulate them but we do not know how they are represented in reality.

First, we present each object separately. Second, we explain why we need a redefinition of the object representing a predicate. Finally, we conclude and state a limitation of the current version of Gödel.

Remark: The words predicate and atom will be used throughout the text to denote the same concept.

3.1 Indexed Variable

An *indexed variable* is composed of two parts:

- a variable (called the *root*);
- an index.

The main manipulation of an indexed variable is called *index replacement.*

Definition: Let X_j be an indexed variable of index j, the *replacement* of an index i by an integer a applied to X_j gives:

- Xa if $i = j$
  where Xa is a variable constructed with the root X of the indexed variable and the integer a;
- X_j if $i \neq j$.

The service *ReIndexIVar* allows index replacement [1] to indexed variables. The scope of the index replacement is restricted to indexed variables having their index equal to the index to replace. There exist services to create an indexed variable, to test if an object is an indexed variable, to obtain the different components of an indexed variable and to create the string representing an indexed variable.

[1] The way of constructing a variable with the root of an indexed variable and an integer is defined explicitly in the specification of the service *ReIndexIVar*.

3.2 Vector of Variables

A problem arising when constructing a schema is the non-fixed number of arguments of a second-order predicate. The purpose of the vector of variables is to represent a finite sequence of variables without a fixed length.

Example: The *Decompose* predicate in our Divide-and-Conquer schema has to represent any kind of decomposition of the induction parameter. The decomposition of the induction parameter varies according to the type of the parameter (it can even vary for induction parameters of the same type). The head-tail decomposition of an induction parameter of type LIST, $decomplist(L, HL, TL)$, has three arguments. The decomposition of an induction parameter of type TREE, $decomptree(T, L, LST, RST)$, has four arguments. We will, thus, define a *Decompose* predicate allowing the representation of any kind of decomposition and having its number of arguments not fixed thanks to vectors of variables. The *Decompose* predicate, $Decompose(X, \mathbf{HX}, \mathbf{TX})$, has two vectors of variables: **HX** and **TX**. These vectors represent a non-fixed number of head arguments and a non-fixed number of tail arguments respectively.

A *vector of variables* is composed of three parts:

- a variable (called the *root* of the vector);
- a lower bound;
- an upper bound.

A *bound* is either a natural number or a variable. The root is ranging between the lower and upper bounds.

Example: $X[1..n]$ is a vector of variables where X is the root, 1 the lower bound and n the upper bound of the vector.

During the synthesis process, we obtain information about the number of arguments of a predicate. It leads to the instantiation of bounds, which are variables, by a natural number.

Definition: a vector is *instantiated* if its lower and upper bounds are, both, natural numbers.

The service *VectorInstantiated* tests if a vector is instantiated. When a vector is instantiated, we can replace it by the sequence of variables ranging between the lower and upper bounds. This replacement is called *expansion* of the instantiated vector.

Definition: Let $X[i..j]$ be an instantiated vector of variables ($i, j \in IN$), the *expansion* of $X[i..j]$ is defined as follows:

- $X_i, X_{i+1}, \ldots, X_j$ if $i < j$;
- X_i if $i = j$;

- the empty sequence if $i > j$.

Remark: An instantiated vector having its lower bound greater than its upper bound will disappear from the arguments of a predicate.
Definition: The length of an instantiated vector $X[i..j]$, denoted $Length(X[i..j])$, is defined as follows:

- $Length(X[i..j]) = j - i + 1$ if $i \leq j$;
- $Length(X[i..j]) = 0$ if $i > j$.

The root of a vector being a variable is too restrictive to represent all possible cases. We will generalize the notion of vector of variables to vectors having their root being a variable, an indexed variable or a vector of variables. We need to extend the previous definitions to this generalized notion of vector.
Definition: Variables and indexed variables are *instantiated*.
A vector is *instantiated* if its lower and upper bounds are natural numbers and if its root is instantiated.
The service *VectorInstantiated* tests if a vector respects this definition.
Definition: Let S be a sequence $E_m, E_{m+1}, \ldots, E_n$,

$$S \leftarrow i = E_{mi}, E_{m+1i}, \ldots, E_{ni}.$$

Definition: Let $X[i..j]$ be an instantiated vector of variables $(i, j \in IN)$ where X is a variable, the *expansion* of $X[i..j]$ is defined as follows:

- $X_i, X_{i+1}, \ldots, X_j$ if $i < j$;
- X_i if $i = j$;
- the empty sequence if $i > j$.

Let $X_k[i..j]$ be an instantiated vector of variables $(i, j \in IN)$ where X_k is an indexed variable, the *expansion* of $X_k[i..j]$ is defined as follows:

- $X_{ki}, X_{ki+1}, \ldots, X_{kj}$ if $i < j$;
- X_{ki} if $i = j$;
- the empty sequence if $i > j$.

Let $V[i..j]$ be an instantiated vector of variables where V is a vector, the *expansion* of $V[i..j]$, denoted $Expand(V[i..j])$ is defined as follows:

- $Expand(V) \leftarrow i, Expand(V) \leftarrow i + 1, \ldots, Expand(V) \leftarrow j$ if $i < j$;

- $Expand(V) \leftarrow i$ if $i = j$;
- the empty sequence if $i > j$.

The service *ExpandVector* expands an instantiated vector.
Example: The expansion of the instantiated vector $X[2..3][1..2]$ is obtained like this:

$$Expand(X[2..3]) \leftarrow 1, Expand(X[2..3]) \leftarrow 2$$

$$(X_2, X_3 \leftarrow 1), (X_2, X_3 \leftarrow 2)$$

$$X_{21}, X_{31}, X_{22}, X_{32}$$

Definition: Let X be a variable, an indexed variable or a vector of variables, the length of X, denoted $Length(X)$, is defined as follows:

- $Length(X) = 1$ if X is a variable or an indexed variable;
- $Length(X) = (Length(R)) \times (j - i + 1)$ if X is a vector $R[i..j]$ and $i \leq j$;
- $Length(X) = 0$ if X is a vector $R[i..j]$ and $i > j$.

Example: The length of the vector $X[2..3][1..2]$ is:

$$(1) \times (3 - 2 + 1) \times (2 - 1 + 1) = 4$$

There exist services to create a vector of variables, to test if an object is a vector of variables, to obtain the different components of a vector and to create the string representing a vector.

3.3 Indexed Predicate

An *indexed predicate* is composed of three parts:

- a predicate variable;
- an index;
- a list of arguments.

Example: $P_k(X_1, \ldots, X_n)$ is an indexed predicate where P is the predicate variable, k the index and $X_1, \ldots, X_n$ the list of arguments of the indexed predicate.
The list of arguments of an indexed predicate contains terms and/or indexed variables and/or vectors of variables.
Definition: The set of terms, indexed variables and vectors of variables is called *schema terms*.

We need to extend the definition of the *index replacement* to schema terms and indexed predicates.

Definition: The *replacement* of the index i by the integer a applied to a schema term T, denoted $Replace_{i \to a}(T)$, is defined as follows:

- $Replace_{i \to a}(T) = T$ if T is a term;
- $Replace_{i \to a}(T) = Xa$ if T is an indexed variable X_j and $i = j$ (where Xa is a variable);
- $Replace_{i \to a}(T) = X_j$ if T is an indexed variable X_j and $i \neq j$;
- $Replace_{i \to a}(T) = (Replace_{i \to a}(R))[lb..ub]$ if T is a vector $R[lb..ub]$.

Definition: Let $P_k(X_1, \ldots, X_n)$ be an indexed predicate, the *replacement* of the index i by the integer a applied to $P_k(X_1, \ldots, X_n)$ gives:

- the second-order predicate $Pa(Y_1, \ldots, Y_n)$ if $i = k$
  where $Y_s = Replace_{i \to a}(X_s)$ $(s = 1..n)$;
- the indexed predicate $P_k(Y_1, \ldots, Y_n)$ if $i \neq k$
  where $Y_s = Replace_{i \to a}(X_s)$ $(s = 1..n)$.

The service *ReIndexIAtom* allows index replacement to indexed predicates.

Example: The replacement of the index k by the integer 3 applied to the indexed predicate $P_k(X_k[2..n], Y_j, 7, W)$ gives the second-order predicate $P3(X3[2..n], Y_j, 7, W)$.

The scope of an index replacement is restricted to indexed predicates and indexed variables having their index equal to the index to replace.

Definition: Let $P_k(X_1, \ldots, X_n)$ be an indexed predicate, the *expansion* of $P_k(X_1, \ldots, X_n)$ gives the indexed predicate $P_k(Y_1, \ldots, Y_m)$ where all the instantiated vectors of variables belonging to the list of arguments $X_1, \ldots, X_n$ have been expanded.

The service *ExpandIAtom* expands an indexed predicate. An indexed predicate represents second-order predicates having equivalent semantics but which will be instantiated with different first-order predicates. There exist services to create an indexed predicate, to test if an object is an indexed predicate, to obtain the different components of an indexed predicate and to create the string representing an indexed predicate.

3.4 Conjunction

A *conjunction* is composed of four parts:

- an AND-formula;

- a lower bound;
- an upper bound;
- an index.

The AND-formula can be constructed with predicates, indexed predicates, their negation and/or conjunctions associated thanks to AND-connectives.
Example: $\bigwedge_{j=lb}^{ub} P_j(L, HL, TL)$ & $Q(X_j, Y)$ is a conjunction where $P_j(L, HL, TL)$ & $Q(X_j, Y)$ is the AND-formula, lb the lower bound, ub the upper bound and j the index of the conjunction.
Remark: A parallelism exists between this object and the mathematical notation of the sum $\sum_{j=1}^{n} x_i y_j$.
The purpose of a conjunction is to represent an AND-formula of a finite but non-fixed length and containing indexed objects (indexed variables and/or indexed predicates). The indexed objects having their index equal to the index of the conjunction are ranging between the lower and upper bounds of the conjunction.
Remark: At least one indexed object belonging to the AND-formula of a conjunction has its index equal to the index of the conjunction. If it is not the case, the conjunction is useless. This idea is, also, present for the mathematical notation of the sum where $\sum_{j=1}^{n} x_i = x_i$. We will only consider conjunctions respecting this idea in the rest of the text.
During the synthesis process, we obtain information about the length of a conjunction. It leads to the instantiation of bounds, which are variables, by a natural number.
Definition: A conjunction is *instantiated* if its lower and upper bounds are, both, natural numbers.
The service *ConjInstantiated* tests if a conjunction is instantiated.
When a conjunction is instantiated, we can replace it by the AND-formula ranging between the lower and upper bounds. This replacement is called *expansion* of the conjunction. Before defining the expansion of a conjunction, we have to introduce the notion of *index replacement* applied to AND-formulas.
Definition: Let R be an AND-formula, the *replacement* of the index i by the integer a applied to R, denoted $FormReplace_{i \to a} R$, gives:

- $P(Replace_{i \to a} T_1, \ldots, Replace_{i \to a} T_n)$ if R is a predicate $P(T_1, \ldots, T_n)$;
- $Pa(Replace_{i \to a} T_1, \ldots, Replace_{i \to a} T_n)$ if R is an indexed predicate $P_k(T_1, \ldots, T_n)$ and $i = k$;
- $P_k(Replace_{i \to a} T_1, \ldots, Replace_{i \to a} T_n)$ if R is an indexed predicate $P_k(T_1, \ldots, T_n)$ and $i \neq k$;

- $\neg(FormReplace_{i\to a}P)$ if R is a formula $\neg(P)$ where P is a predicate or an indexed predicate;
- $\bigwedge_{j=lb}^{ub}(FormReplace_{i\to a}A)$ if R is a conjunction $\bigwedge_{j=lb}^{ub}A$;
- $FormReplace_{i\to a}P$ & $FormReplace_{i\to a}Q$ if R is a formula P & Q.

Definition: Let $\bigwedge_{j=lb}^{ub}R$ be an instantiated conjunction ($lb, ub \in IN$), the *expansion* of the conjunction is defined as follows:

- $FormReplace_{j\to lb}(R)$ & $FormReplace_{j\to lb+1}(R)$ & ... & $FormReplace_{j\to ub}(R)$ if $lb < ub$;
- $FormReplace_{j\to lb}(R)$ if $lb = ub$;
- $True$ if $lb > ub$.

The service *ExpandConj* expands an instantiated conjunction.
Example: The expansion of the instantiated conjunction

$$\bigwedge_{j=1}^{2} Pred_j(TX_j, TY) \;\&\; Atom(W_i)$$

is obtained like this:

$$FormReplace_{j\to 1}(Pred_j(TX_j, TY) \;\&\; Atom(W_i))$$

$$\&\; FormReplace_{j\to 2}(Pred_j(TX_j, TY) \;\&\; Atom(W_i))$$

$$Pred_1(TX_1, TY) \;\&\; Atom(W_i) \;\&\; Pred_2(TX_2, TY) \;\&\; Atom(W_i)$$

There exist services to create a conjunction, to test if an object is a conjunction, to obtain the different components of a conjunction and to create the string representing a conjunction.

3.5 Disjunction

A *disjunction* is composed of four parts:

- an AND-formula;
- a lower bound;
- an upper bound;
- an index.

The AND-formula can be constructed with predicates, indexed predicates, their negation and/or conjunctions associated thanks to AND-connectives.
Example: $\bigvee_{j=lb}^{ub} P_j(L, HL, TL)$ & $Q(X_j, Y)$ is a disjunction where $P_j(L, HL, TL)$ & $Q(X_j, Y)$ is the AND-formula, lb the lower bound, ub the upper bound and j the index of the disjunction.
The purpose of a disjunction is similar to the purpose of a conjunction. A disjunction allows the representation of an OR-formula (AND-formulas associated thanks to OR-connectives) of a finite but non-fixed length and containing indexed objects (indexed variables and/or indexed predicates). The indexed objects having their index equal to the index of the disjunction are ranging between the lower and upper bounds of the disjunction.
Remark: At least one indexed object belonging to the AND-formula of a disjunction has its index equal to the index of the disjunction. If it is not the case, the disjunction is useless. We will only consider disjunctions respecting this idea in the rest of the text.
During the synthesis process, we obtain information about the length of a disjunction. It leads to the instantiation of bounds, which are variables, by a natural number.
Definition: A disjunction is *instantiated* if its lower and upper bounds are, both, natural numbers.
The service *DisjInstantiated* tests if a disjunction is instantiated.
Definition: Let $\bigvee_{j=lb}^{ub} R$ be an instantiated disjunction ($lb, ub \in IN$), the *expansion* of the disjunction is defined as follows:

- $FormReplace_{j \to lb}(R) \vee FormReplace_{j \to lb+1}(R) \vee \ldots \vee FormReplace_{j \to ub}(R)$ if $lb < ub$;
- $FormReplace_{j \to lb}(R)$ if $lb = ub$;
- $False$ if $lb > ub$.

The service *ExpandDisj* expands an instantiated disjunction. There exist services to create a disjunction, to test if an object is a disjunction, to obtain the different components of a disjunction and to create the string representing a disjunction.

3.6 Redefinition of Predicate

The object representing a predicate is already defined in the module **Syntax** provided by Gödel. We need to redefine this object to allow the representation of *first-order predicates* and *second-order predicates.*
A *first-order predicate* is composed of two parts:

- a predicate symbol;
- a list of terms.

The service *IsInstAtom* tests if an object is a first-order predicate.
A *second-order predicate* is composed of two parts:

- a predicate variable;
- a list of schema terms.

A second-order predicate can be instantiated by a schema substitution to a first-order predicate (or even a formula). The instantiation of all the second-order predicates of a schema with adequate first-order predicates is the main purpose of a synthesis.
Definition: Let $P(X_1, \ldots, X_n)$ be a second-order predicate,
the *expansion* of $P(X_1, \ldots, X_n)$ gives the second-order predicate $P(Y_1, \ldots, Y_m)$ where all the instantiated vectors of variables belonging to the list of arguments $X_1, \ldots, X_n$ have been expanded.
The service *ExpandAtom* expands a second-order predicate. There exist services to create a predicate, to test if an object is a predicate, to obtain the different components of a predicate and to create the string representing a predicate.

3.7 Conclusion

We have seen the different objects available in the module **Sobjects** together with the services allowing their creation and manipulation. By gathering these schema objects with objects allowing the representation of terms and formulas, we can now construct schemas. The services presented in this section are the basic blocks for constructing new services allowing the manipulation of schemas. These new services are described in the next section.
Remark: If we examine the arguments of most of the services belonging to the module **Sobjects**, we see that we need the representation of a Gödel program. This is due to a feature of the current version of Gödel. Gödel offers a facility called *overloading*. It allows the definition of symbols (types, constants, functions, propositions and predicates) with the same name. The only condition is the following: distinct symbols cannot be declared in the same module with the same category, name and arity. This facility allows us to define, for example, the addition of two reals and the addition of two integers with the same name: +. To identify a symbol, we need more information than only its name. We also need the module where the symbol has been declared, the category of the symbol and its arity. The identifier of the constant *Nil* is the tuple $(NAME : Nil, MODULE : Lists, CATEGORY : Constant, ARITY : 0)$. An identifier is a complex object and its manipulation is restricted. There are no services to construct an identifier for a symbol. The only service available is the extraction of the name we need from the representation of a Gödel program. An argument of type PROGRAM is present in most of the services of the module **Sobjects** to allow the extraction of the names needed.

Chapter 4

Services Offered by the Module Schemas

We have defined the notion of schema and we will now focus on services available to manipulate schemas. These services are not specific to a particular schema like Divide-and-Conquer, Generate-and-Test, ... They can, thus, be used with any schema respecting the framework defined in the introduction, that is to say, a schema has to be of the form $Head \Leftrightarrow Body$ where *Head* is an atom and *Body* a formula in disjunctive normal form and a schema has to be constructed with objects belonging to the modules **Syntax** and/or **Sobjects**. We will present services which give the string representing a schema, which create the program corresponding to a schema in a target language and which translate the type SCHEMA into the type FORMULA (and vice versa). The service allowing the instantiation of a schema is described in the next section.

4.1 String Representation

The service *SchemaToString* gives the representation of a schema with a string of characters. It facilitates the display of a schema on screen and its printing. Each object composing a schema has to be represented in an unambiguous manner.
Example: This is the representation of the Divide-and-Conquer schema when using the service *SchemaToString*:

```
R(X,Y)  <->
        Minimal(X)
        & Solve(X,Y)
OR      NonMinimal(X)
        & Decompose(X,HX[1..hx],TX[1..t])
        AND[1=<j=<t] R(TXj,TYj)
        & Process(HX[1..hx],HY[1..hy])
        & Compose(HY[1..hy],TY[1..t],Y)
```

The representation of a schema is possible at any state of instantiation.

The service *InstSchemaToString* gives the representation of the instantiated part of a schema with a string of characters.

Definition: The instantiated part of a schema is composed of the instantiated predicates belonging to the schema and the logical AND/OR-connectives keeping the structure of the schema for these predicates.

Example: This is the representation, when using the service *InstSchemaToString*, of the Divide-and-Conquer schema where its predicates R, $Minimal$, $NonMinimal$ and *Decompose* have been instantiated:

```
sort(X,Y)  <->
           X = []
OR         Not(X = [])
           & X = [HX|TX]
```

This service allows a step-by-step view of the synthesis process.

4.2 Program Derivation

An executable program can be derived from a schema resulting of the synthesis process. The services *SchemaToGoedel* and *SchemaToProlog* create the representation of the programs corresponding to an instantiated schema in the target languages Gödel and Prolog respectively.

Definition: An instantiated schema is a schema where all its predicate variables and form variables have been instantiated.

The service *InstantiatedSchema* tests if a schema respects this definition. The representation of the programs obtained thanks to these services are stored in files. Only Gödel and Prolog are supported but it is easy to add a service which can create the representation of a program for another language. An instantiated schema can be seen as a logic algorithm and its derivation into an executable program as a mechanical process (for the definition of logic algorithm, see [2]).

Remark: The derivation of a program from an instantiated schema is reduced to a purely

syntactical process in this case and does not correspond to the process defined in [2] involving complex manipulations like type checking, directionality checking and multiplicity checking.

4.3 Types Translation

The service *FormulaSchema* allows to obtain the formula corresponding to a schema and vice versa. Pragmatic reasons have guided the construction of this service. It is not possible in Gödel to redefine the name of a type (to give a synonym for a type). We used a constructor to enclose a formula and then obtain a schema. But, by doing this, we can't use the facilities offered for formulas anymore. The service *FormulaSchema* allows to use the facilities offered for both formulas and schemas.
The specifications of all these services are in the appendix.

Chapter 5

Schema Substitution

A *schema substitution* is an object allowing the representation of information to instantiate a *schema*. The application of a schema substitution to a schema instantiates placeholders in the schema. Instantiating all the placeholders of a schema leads to the desired synthesized *logic algorithm*. First, we give an introduction of the notion of schema substitution and its objective in the abstract data type Schema. Second, we describe the different kinds of bindings belonging to a schema substitution. Third, we state the restrictions of a schema substitution compared to a term substitution. Finally, we conclude and give an overview of the implementation of the schema substitution using the facilities offered by Gödel.

5.1 Introduction

A *schema substitution* is a finite sequence of bindings. There are three different kinds of objects that can be instantiated in a schema:

- first-order variables;
- second-order variables (also called predicate variables);
- form variables.

There are three different kinds of bindings in a schema substitution corresponding to these three objects:

- variable/term binding;
- predicate variable/λ-term binding;
- form variable/integer binding.

A binding contains information to instantiate a schema. This information is provided by methods. Briefly, a *method* receives a part of the specification of the desired algorithm to synthesize, a predicate variable and gives information to instantiate this predicate variable and possibly to instantiate variables and form variables. After the instantiation of all the variables, predicate variables and form variables of a schema thanks to different methods, we obtain, from the specification, the desired synthesized logic algorithm. The objective met by the notion of schema substitution is offering an object unifying the representation of the results given by different methods and allowing the instantiation of different objects in a schema. The schema substitution supports the essential services of the abstract data type Schema.

5.2 Kinds of Bindings

A schema substitution is composed of three kinds of bindings. We will present each binding separately.

5.2.1 Variable/Term Binding

The variable/term binding is the binding of the classical *term substitution*. Let $\theta = \{V_1/T_1, V_2/T_2, \ldots, V_n/T_n\}$ be a schema substitution where the V_i/T_i are variable/term bindings ($i = 1..n$) and S be a schema. Then the application of θ to S (noted Sθ) gives the schema S' which is obtained from S by simultaneously replacing each occurrence of the variable V_i in S by the term T_i ($i = 1..n$). A schema substitution strictly composed of variable/term bindings can be seen as a term substitution where its application is restricted to schemas (for a detailed description of the term substitution, see [8]).

5.2.2 Predicate Variable/λ-Term Binding

A predicate variable is a second-order variable. The instantiation of a predicate variable is more complex than a substitution of predicate symbols. Throughout this paper, a predicate variable will start with an uppercase letter and a predicate symbol will start with a lowercase letter.

Example: the Generate-and-Test schema

$$R(X,Y) \Leftrightarrow Generate(X,Y)\ \&\ Test(Y)$$

leads to the logic algorithm *sort/2*

$$sort(X,Y) \Leftrightarrow permutation(X,Y)\ \&\ ordered(Y)$$

via the substitution predicate variable/predicate symbol

$$\{R/sort, Generate/permutation, Test/ordered\}$$

This approach is not satisfactory because we need more powerful facilities like the permutation of arguments in a predicate, the elimination of useless arguments, the introduction of new arguments and the instantiation of a predicate variable by a formula. The notion of λ-terms allows us to obtain these facilities.

Definition: Assume that there is given an infinite set of distinct symbols called variables, and an infinite set of distinct symbols called constants. The set of expressions called λ-terms is defined inductively as follows:

- All variables and constants are λ-terms (called atoms);
- If M and N are any λ-terms, then (MN) is a λ-term (called a λ-application);
- If M is any λ-term and x is any variable, then $(\lambda x.M)$ is a λ-term (called an abstraction).

Definition: A restricted abstraction is an abstraction $(\lambda x.M)$ where M is an atom, a λ-application not containing an abstraction or a restricted abstraction.

Example: $(\lambda x.(\lambda y.(xy)))$ is a restricted abstraction but $(\lambda x.(z(\lambda w.w)))$ is not a restricted abstraction.

In the following, we will only use restricted abstractions.

Definition: The number of arguments of a λ-term T, noted $NbArg(T)$, is defined inductively as follows:

- $NbArg(T) = 0$ where T is an atom or a λ-application;
- $NbArg(\lambda x.M) = 1 + NbArg(M)$ where $\lambda x.M$ is a restricted abstraction.

Definition: Let T be a λ-term containing an occurrence of the form $((\lambda x.M)N)$, the process of replacing this occurrence by $M\{x/N\}$ in T (where $\{x/N\}$ is a term substitution) is called reduction. The application of a schema substitution composed of a predicate variable/λ-term binding to a schema uses this notion of reduction.

Example: The application of the schema substitution

$$\{Pred/\lambda A.A = []\}$$

to the schema

$$R(X, Y) \Leftrightarrow Pred(X)$$

gives the following reduction

$$R(X, Y) \Leftrightarrow ((\lambda A.A = [])(X))$$

$$R(X,Y) \Leftrightarrow (A = []\{A/X\})$$
$$R(X,Y) \Leftrightarrow X = []$$

Remark: The number of arguments of the predicate *Pred* has to be equal to the number of arguments of the λ-term (for a detailed description of the notion of λ-term, see [7], and for the idea of using λ-terms in logic programming, see [9]).
Remark: The services to create and manipulate λ-terms are gathered in the module **Sobjects**.
We will now examine in detail the facilities obtained by the use of λ-terms.

Permutation of Arguments.

The instantiation of a predicate variable by a λ-term allows us to permute the order of the arguments in the predicate.
Example: The Divide-and-Conquer schema

$$\begin{aligned} R(X,Y) \Leftrightarrow \\ \quad & Minimal(X) && \& \quad Solve(X,Y) \\ \vee \quad & NonMinimal(X) && \& \quad Decompose(X, \mathbf{HX}, \mathbf{TX}) \\ & && \& \quad \mathbf{R}(\mathbf{TX}, \mathbf{TY}) \\ & && \& \quad Process(\mathbf{HX}, \mathbf{HY}) \\ & && \& \quad Compose(\mathbf{HY}, \mathbf{TY}, Y) \end{aligned}$$

induces that the induction parameter is X, the first argument of R. If we want to synthesize an algorithm for the predicate $member(E, L)$ where E is a term and L is a list of terms, we have to permute the order of the arguments in R, the induction parameter L being the second argument. The application of the schema substitution

$$\{R/\lambda A.\lambda B.member(B, A)\}$$

to our Divide-and-Conquer schema will give the following schema

$$\begin{aligned} member(Y,X) \Leftrightarrow \\ \quad & Minimal(X) && \& \quad Solve(X,Y) \\ \vee \quad & NonMinimal(X) && \& \quad Decompose(X, \mathbf{HX}, \mathbf{TX}) \\ & && \& \quad \mathbf{member}(\mathbf{TY}, \mathbf{TX}) \\ & && \& \quad Process(\mathbf{HX}, \mathbf{HY}) \\ & && \& \quad Compose(\mathbf{HY}, \mathbf{TY}, Y) \end{aligned}$$

This facility allows not only a permutation of two arguments but a complete reorder of the arguments in a given n-ary predicate.
Remark: In the following examples, we will always apply a schema substitution to this Divide-and-Conquer schema.

Elimination of Useless Arguments.

The instantiation of a predicate variable by a λ-term allows us to eliminate useless arguments in the predicate. This elimination is also called projection.
Example: the schema substitution

$$\{Compose/\lambda A.\lambda B.\lambda C.B = C\}$$

applied to our previous Divide-and-Conquer schema (where TX and TY are not vectors of variables but simply variables) leads to the schema

$$R(X,Y) \Leftrightarrow$$

$$\begin{array}{rrl} & Minimal(X) & \& \quad Solve(X,Y) \\ \vee & NonMinimal(X) & \& \quad Decompose(X, \mathbf{HX}, TX) \\ & & \& \quad R(TX,TY) \\ & & \& \quad Process(\mathbf{HX}, \mathbf{HY}) \\ & & \& \quad TY = Y \end{array}$$

The argument HY has disappeared in the instantiation of the predicate *Compose*. The number of arguments of the predicate $Compose(\mathbf{HY}, TY, Y)$ is equal to the number of arguments of the λ-term $\lambda A.\lambda B.\lambda C.B = C$.

Introduction of New Arguments.

Similarly, it is possible to introduce new arguments when instantiating a predicate variable by a λ-term.
Example: the schema substitution

$$\{Minimal/\lambda A.length(A,0)\}$$

applied to our Divide-and-Conquer schema leads to the following schema

$$R(X,Y) \Leftrightarrow$$

$$\begin{array}{rrl} & length(X,0) & \& \quad Solve(X,Y) \\ \vee & NonMinimal(X) & \& \quad Decompose(X, \mathbf{HX}, \mathbf{TX}) \\ & & \& \quad \mathbf{R}(\mathbf{TX}, \mathbf{TY}) \\ & & \& \quad Process(\mathbf{HX}, \mathbf{HY}) \\ & & \& \quad Compose(\mathbf{HY}, \mathbf{TY}, Y) \end{array}$$

The predicate length is binary as opposed to Minimal which is unary.
Remark: When introducing new variables in a predicate using this facility, we must keep in mind that these variables can interfere with variables already in the schema. Introducing a new variable X in a Divide-and-Conquer schema will blur the distinction between this one and the variable X representing the induction parameter.

Instantiation of a Predicate Variable by a Formula.

In the previous examples, we have seen the possibilities of instantiating a predicate variable by a predicate where the arity of this predicate and the position of its arguments can be different from the predicate variable. We are not restricted to predicates only. It is also possible to instantiate a predicate variable by a formula.
Example: the schema substitution

$$\{NonMinimal/\lambda A.length(A, Lg)\ \&\ Lg > 0\}$$

applied to our Divide-and-Conquer schema leads to the following schema

$$\begin{array}{rlll} & & R(X,Y) \Leftrightarrow & \\ & Minimal(X) & \& & Solve(X,Y) \\ \vee & length(X,Lg) & \& & Lg > 0 \\ & & \& & Decompose(X, \mathbf{HX}, \mathbf{TX}) \\ & & \& & \mathbf{R}(\mathbf{TX}, \mathbf{TY}) \\ & & \& & Process(\mathbf{HX}, \mathbf{HY}) \\ & & \& & Compose(\mathbf{HY}, \mathbf{TY}, Y) \end{array}$$

We restricted the formula in a λ-term to be in disjunctive normal form. The possibility of having an OR-connective (or several) in a λ-term will lead to the normalization of the schema being instantiated. The normalization is necessary because we want to respect the definition of a schema (its body being a formula in disjunctive normal form).
Remark: Representing a formula with λ-terms is possible. The logical connectives belong to the set of constants (for a complete discussion, see [9]).

Argument Name Independence.

To facilitate the construction of an abstraction, we have to ensure an important property: *the argument name independence.* When constructing an abstraction to bind with a predicate variable, we don't want to bother about the representation of the arguments of the predicate to instantiate.
Example: the application of the schema substitution

$$\{Pred/\lambda B.\lambda C.\lambda A.B = C\}$$

to a schema S containing the predicate $Pred(A, B, C)$ has to result in a schema S' which is obtained from S by simultaneously replacing all the occurrences of *Pred* by $A = B$. Because of name clashes, we obtain in fact a schema S" which is obtained from S by simultaneously replacing all the occurrences of *Pred* by $C = B$. The desired elimination of the third argument results in fact in the elimination of the first argument. This problem has been solved with an automatic renaming of the arguments of the abstraction causing clashes before the application.

5.2.3 Form Variable/Integer Binding

During the synthesis of a logic algorithm, we obtain information about the numbers of arguments for a predicate, the number of recursive calls,... This information allows us to precise the final form of a schema and is represented using the form variable/integer binding. In our Divide-and-Conquer schema, the form variables are hx, hy and t. They are the upper bounds of the vectors of variables and of the conjunction appearing in the Divide-and-Conquer schema. When instantiating a predicate variable, a method gives also, implicitly or explicitly, information to instantiate form variables.

Example: From the choice of instantiating *Decompose* with the head-tail decomposition (assuming that the induction parameter X is a list) $X = [HX|TX]$, we can extract the fact that the numbers of heads and tails of X will be equal to one. We obtain, thus, this schema substitution $\{hx/1, t/1\}$.

Example: The application of the schema substitution

$$\{hx/1, t/2, Decompose/\lambda T.\lambda L.\lambda LST.\lambda RST.T = Tree(L, LST, RST)\}$$

to our Divide-and-Conquer schema leads to a schema

$$R(X,Y) \Leftrightarrow$$

$$\begin{array}{rlll} & Minimal(X) & \& & Solve(X,Y) \\ \vee & NonMinimal(X) & \& & X = Tree(HX, TX1, TX2) \\ & & \& & R(TX1, TY1) \\ & & \& & R(TX2, TY2) \\ & & \& & Process(HX, HY) \\ & & \& & Compose(HY, TY1, TY2, Y) \end{array}$$

Knowing that the induction parameter X is decomposed into one head and two tails automatically leads to two recursive calls, one for each tail.

5.3 Restrictions of a Schema Substitution

The restrictions of a schema substitution compared to a term substitution are the order of the bindings in a substitution and the composition of substitutions.

A schema substitution is not a set of bindings as for a term substitution but a sequence of bindings. A schema substitution is composed of different kinds of bindings. Let θ be a schema substitution, τ be the schema substitution θ where two bindings (or more) have been permuted and S be a schema, we can't assure that

$$S\theta = S\tau$$

in general.
Example: The schema substitution

$$\{t/2, Decompose/\lambda T.\lambda L.\lambda LST.\lambda RST.T = Tree(L, LST, RST), hx/1\}$$

when applied to our Divide-and-Conquer schema S will lead to a schema where *Decompose* has been instantiated. If we decide to permute the two first bindings, the application of this schema substitution to S will fail (the number of arguments of the *Decompose* predicate being different from the number of arguments of the abstraction).

The composition of schema substitutions is more delicate than the composition of term substitutions because a schema substitution gathers different kinds of bindings. We must ensure that the bindings will not mix each other.
Example: The composition of the schema substitutions $\{Minimal/\lambda J.J = []\}$ and $\{J/2\}$ does not have to lead to $\{Minimal/\lambda 2.2 = [], J/2\}$.
The problem of the composition has not been tackled completely. A service of concatenation of schema substitutions is supported.

5.4 Conclusion

We have seen the notion of schema substitution, the three kinds of bindings that can appear and the restrictions compared to the classical term substitution. This object has been implemented in Gödel. The objective of reusing the facilities offered by the language has guided the construction of this object. A schema substitution is represented as a term substitution $\{V_1/T_1, V_2/T_2, \ldots, V_n/T_n\}$. The first-order variables, second-order variables and form variables are represented with the variables of Gödel in ground representation. The term T_i is encapsulated in a constructor showing the kind of binding represented ($i = 1..n$). The application of the schema substitution is done sequentially (binding after binding) and differs according to the kind of binding.

Chapter 6

Conclusion and Future Work

We have defined, in this part, the notion of schema and stated the objective this notion is fulfilling in the synthesis. Then, we have presented the schema objects needed to construct complex logic algorithm schemas. Finally, we have listed the different services available to manipulate schemas and to instantiate them via schema substitutions. The abstract data type representing logic algorithm schemas is one of the basic blocks of the architecture presented in the next part.

There are several ways of improving the abstract data type representing logic algorithm schemas defined in this part.

First, we can relax the restriction stated for bounds, that is to say, a bound is either a natural number or a variable. We can allow a bound to be an arithmetic expression too. The effect of this improvement is that it takes care of specific constraints.

Example: The following schema

$$R(X,Y) \Leftrightarrow$$

$$\ldots$$

$$\begin{array}{lll} \bigvee_{j=1}^{k} & NonMinimal(X) & \& \ \ldots \\ \bigvee_{j=k+1}^{l} & NonMinimal(X) & \& \ \ldots \end{array}$$

expresses that there exists a relation between the upper bound of the first disjunction and the lower bound of the second one. The current implementation only allows this schema to be represented using a variable k' for the lower bound of the second disjunction instead of the arithmetic expression $k+1$. The constraint $k+1=k'$ has to be ensured "manually" when instantiating k and k'. By allowing arithmetic expressions for bounds, we can represent the schema above easily. The instantiation of k automatically respects the constraint existing between the two bounds. Another advantage of this solution is that it only requires the instantiation of the variable k instead of two different instantiations (one for k and another for k').

Second, the schema substitution is a sequence of bindings and not a set as for a classical term substitution. The application of a schema substitution is order-sensitive. We have,

thus, to ensure that the different bindings are in a certain order to obtain the result of the application we desired. A solution to remove this inconvenience is to define an order for the bindings of a schema substitution and to sort the schema substitution given to obtain the one we will finally apply.

Remark: The definition of an order for the bindings is not a trivial task. The sorting is dependent on the kind of schema being instantiated and also on other information.

Example: If a second-order predicate $Pred(\mathbf{X}, \mathbf{Y})$ has to be instantiated by the following schema substitution:

$$\{\ldots, Pred/\lambda A.\lambda B.\lambda C.split(A, B, C), \ldots\}$$

where $\mathbf{X}$ and $\mathbf{Y}$ represent respectively the vectors of variables $X[1..hx]$ and $Y[1..hy]$, we can observe that the number of arguments of the second-order predicate is different from the number of arguments of the λ-term. We know that a form variable/integer binding has to be placed before the predicate variable/λ-term binding. But if the schema substitution contains the bindings $hx/2$ and $hy/2$, we still don't know which one of these two has to be placed before the predicate variable/λ-term binding and which one has to be placed after.

Part II

A Modular Architecture for an Inductive Schema-Guided Logic Program Synthesizer

The architecture for an inductive schema-guided logic program synthesizer called **DIALOGS**, presented in this section, is composed of five main blocks:

- the *specification* block;
- the *schemas creation* block;
- the *methods* block;
- the *strategies* block;
- the *synthesizer* block.

The objective guiding the construction of this modular architecture is the following: the architecture has to allow the definition and the addition of new methods, new schemas, new strategies and even new specification frameworks without a redesign of the synthesizer. The synthesizer is viewed as a workbench with a disparate toolbox containing highly specialized methods for a set of schemas.

First, we present each block separately and illustrate their content with an example corresponding to the **SYNAPSE** synthesizer defined in [4]. Second, we explain how to define and add new objects to each block to improve the synthesizer constructed in the previous section. Finally, we conclude and introduce this synthesizer as a complete tool in a logic programs development environment.

Chapter 7

Description of the Blocks Composing the Architecture

First, we give the insight of how the synthesizer is working and the role of the blocks proposed. Then, we define each block composing the architecture of the inductive schema-guided logic program synthesizer **DIALOGS**.

7.1 Overview of the Synthesizer

We will now give an idea on how the synthesizer is working. First, the user constructs a specification for a predicate p in a particular framework. Then, he can choose a schema (or let the synthesizer choose a schema) that will be instantiated to obtain a logic algorithm for the predicate p. The user can choose (or let the synthesizer choose) in the strategies related to the schema used for the synthesis, the one to be applied. The strategy is a sequence of steps instantiating the placeholders of a schema by using different methods provided by a toolbox. After the application of the strategy, the synthesizer shows the resulting logic algorithm for the predicate p to the user. The user can restart a synthesis with another choice for the schema and/or the strategy or with a modification of his specification for the predicate p. If the resulting logic algorithm is what the user wants, he can obtain an executable code from this algorithm in a target language.

7.2 Specification Block

The synthesis of a logic algorithm for a predicate p requires a specification of this predicate. The module **Specifs** provides a framework allowing the definition of predicates to be synthesized. It also provides services to extract specifications from a file (or from the standard input stream) and services to manipulate specifications.

A possible framework for specifications is composed of:

- the predicate symbol and the arguments of the predicate for which the user wants a logic algorithm;
- the types of the arguments of this predicate;
- a set of positive examples for which the predicate has to succeed;
- a set of axioms of this predicate.

Remark: This framework is the same as the one defined in [4].
Example: Here is a specification of the predicate $member/2$ in the framework defined above:

$$member(E, L)$$

$$E : Term$$
$$L : List(Term)$$

$$member(a, [a])$$
$$member(b, [a, b])$$

$$member(X, L) \Leftarrow append(Pre, [X|Post], L)$$

where the predicate $append(A, B, C)$ is considered as a primitive and can be informally defined like this: C is the concatenation of B to the end of A.

The export part of the module **Specifs** is constructed like this:

```
EXPORT Specifs.

IMPORT Sobjects, Examples.

BASE Specif.

PREDICATE    DialogConstructSpecif:
             Specif.                 The representation of a
                                     specification extracted from the
                                     standard input via a dialog with
                                     the user.
```

```
PREDICATE     ConstructSpecif:
              String                 A string.
            * Specif.                The representation of a
                                     specification extracted from a
                                     file having its root name equal
                                     to the first argument and its
                                     extension equal to ''spf.''
PREDICATE     ComponentsOfSpecif:
              Specif                 The representation of a
                                     specification.
            * String                 The specified predicate.
            * List(Type)             The list of types of the
                                     arguments of the specified
                                     predicate.
            * List(Example)          A list of positive examples for
                                     which the specified predicate
                                     has to succeed.
            * List(Formula).         A list of axioms of the
                                     specified predicate.
PREDICATE     SpecifToString:
              Specif                 The representation of a
                                     specification.
            * String.                A string representing this
                                     specification.
```

The keyword EXPORT indicates that we are in the export part of the module of name **Specifs**. The keyword IMPORT indicates which services are available to the module **Specifs** via the modules **Sobjects** and **Examples** (where the module **Sobjects** is the one defined in the previous part and where the module **Examples** contains the abstract data type EXAMPLE and services to manipulate this abstract data type). The keyword BASE indicates which types are declared in this module. The type declared is Specif and it is the type of a term representing a specification. The keyword PREDICATE indicates which predicates are declared in the module **Specifs** together with the types of the arguments of each predicate.

Remark: The type, function and predicate symbols start with an uppercase letter to respect the convention of Gödel.

The local part of the module **Specifs** is constructed like this:

```
LOCAL Specifs.

FUNCTION Spcf:  String * List(Type) * List(Example) * List(Formula)
                  -> Specif.

DialogConstructSpecif(spec)                          <-  ...
ConstructSpecif(fname,spec)                          <-  ...
ComponentsOfSpecif(spec,pred,ltype,lex,lform)        <-  ...
SpecifToString(spec,str)                             <-  ...
```

The keyword FUNCTION indicates which functions are declared in the local part of this module. The purpose of the function Spcf is to give the type Specif to terms representing a specification. The local part of the module **Specifs** also contains the definition of the four predicates declared in its export part.

We can see that the module **Specifs** is simple and can use the facilities offered by other modules to manipulate specific objects composing a specification like Type, Example and Formula.

7.3 Schemas Creation Block

The module **Sobjects** and **Schemas** have been described in the previous part. They provide, respectively, services to create and manipulate schema objects and to manipulate schemas. The module **SchemasCreation** gathers the service *CreateSchema* which allows the creation of specific schemas.

The export part of the module **SchemasCreation** is constructed like this:

```
EXPORT SchemasCreation.

IMPORT Schemas.

PREDICATE     CreateSchema:
              String          A string.
            * Schema.         The representation of the schema having
                              its name equal to the first argument.
```

The local part of the module **SchemasCreation** is constructed like this:

```
LOCAL SchemasCreation.

CreateSchema(''DC1'',sch)  <-  ...
CreateSchema(''GT'',sch)   <-  ...
CreateSchema(''DC2'',sch)  <-  ...
```

Remark: Six different Divide-and-Conquer schemas have already been constructed

and are available in the module **SchemasCreation**.
The service *CreateSchema* allows a direct access to the representation of a particular schema if its first argument is instantiated to a specific string. If its first argument is not instantiated, the service *CreateSchema* will iterate on all the representations of schemas contained in the module **SchemasCreation**. This feature allows the synthesizer to give different logic algorithms starting from a single specification or to obtain the best suited schema to synthesize a logic algorithm.

7.4 Methods Block

A *method* extracts, from a specification and possibly from background knowledge, information to instantiate placeholders of a schema. The simplest method existing is the method extracting, from the specification, the predicate symbol for which the synthesizer tries to produce a logic algorithm. A schema substitution is, then, created and binds the predicate variable of the head of the schema to the predicate symbol extracted (which will be under the form of a λ-term). We just have to apply this schema substitution to the schema to instantiate a placeholder.
Example: The method A extracts from the specification, proposed in section 7.2, the predicate $member(E, L)$. A schema substitution $\theta = \{R/\lambda L.\lambda E.member(E, L)\}$ is created and applied to our Divide-and-Conquer schema.
There exist more sophisticated methods that perform actual computations for inferring information. Two methods, called the *Most Specific Generalization* method and the *Proofs-as-Programs* method, are explained in [4].
Remark: To be useful, a method must be independent from a specific schema. However, the construction of a schema substitution is dependent on the schema chosen. The schema substitution has to know the predicate variable of the head of the schema (R in the upper example).
There are five distinct phases in the use of a method:

1. selection of the useful part of the specification for a particular method;
2. use of the method with several arguments (including a part of the specification) to obtain information;
3. test to verify if the information given by the method is acceptable;
4. transformation of the information into a schema substitution;
5. application of the schema substitution obtained to a schema.

The export part of the module **Methods** is constructed like this:

```
EXPORT Methods.

IMPORT Schemas, Specifs.

PREDICATE  MethodA:  TArg1 * TArg2 * TArg3;
           MethodB:  TArg1 * TArg2;
           ...
```

The local part of the module **Methods** contains the definitions of the predicates declared in its export part.

7.5 Strategies Block

A *strategy* is a sequence of steps guiding the complete instantiation of a particular schema S. Each step is a mapping between a method and a placeholder of the schema S. The result given by a method is information to instantiate a placeholder. When all the placeholders of S have been instantiated, the synthesis is over and results in a logic algorithm.
Example: Here is a possible strategy for our Divide-and-Conquer schema:

1. instantiation of R using the method A;
2. instantiation of $Minimal$ using the method B;
3. instantiation of $NonMinimal$ using the method B;
4. instantiation of $Decompose$ using the method C;
5. instantiation of $Solve$ using the method D;
6. ...

Remark: We can imagine another mapping between methods and placeholders or a reorder of the different steps of the strategy above. This leads to the creation of a new strategy for the same schema. A schema can be instantiated by different strategies but a strategy instantiates one and only one schema.
A strategy is represented under the form of a Gödel program instead of being represented under the form of a sequence of couples indicating which placeholder is instantiated thanks to a particular method. This approach allows us to reuse the facilities already available to manipulate Gödel programs and to have a way to define more powerful strategies. A strategy can be more complex than just a sequence of methods/placeholders couples. It is

sometimes important to test the result given by a method. If the result is not the one expected, it must be possible to instantiate the placeholder with another method. A module will be created for each strategy we want to add to the synthesizer.
The export part of a module describing a strategy is constructed like this:

```
EXPORT StrategyName.

IMPORT Specifs, SchemasCreation, Methods.

PREDICATE     Strategy:
              Schema      The representation of a schema.
            * Schema.     The schema corresponding to the first
                          argument where all its predicate variables
                          and form variables have been instantiated.
```

Remark: Each strategy is a module having a name different from the other strategy module names belonging to the synthesizer. But the predicate *Strategy*/2 is always present in all the strategy modules.
The local part of a module describing a strategy is constructed like this:

```
LOCAL StrategyName.

Strategy(schema,result)       <-

GetSpecifFileName(fname)       &
ConstructSpecif(fname,spec)    &
MethodA(_,_,_)                 &
...
```

Remark: A strategy has to take care of the extraction of the specification thanks to the services available in the module **Specifs**. This is due to the fact that we want to have the possibility to add new specification frameworks to the synthesizer. This feature will be explained later.
A strategy is a complex object but its construction can be simplified with a huge and well-designed library of methods.

7.6 Synthesizer block

We already have services to extract and manipulate specifications, services to create and manipulate schemas. We have strategies to guide the instantiation of a schema thanks to methods. The only object still needed is a tool executing the instantiation steps described in a strategy to an adequate schema. This tool is a Gödel interpreter and is the central part of the synthesizer. The interpreter receives a strategy represented as a Gödel program

and the goal

$$CreateSchema(\textbf{schemaname}, schema)\ \&\ Strategy(schema, result)$$

and gives as a result an answer substitution containing the binding $result/algorithm$.
Remark: In the goal above, *schema* and *result* are variables that will be present in the answer substitution. But **schemaname** is not a variable. It is a string representing the name of a particular schema. When a specific strategy is given to the interpreter, it has to instantiate the schema linked to this strategy.
Starting from a Gödel interpreter, we incrementally develop and explain the body of the synthesizer **DIALOGS**.
Remark: An interpreter called *MySucceed* is already available from a meta-programming module of Gödel.

First, the service *StrategyInterpreter* is built above a classical Gödel interpreter.
The export part of the module **Dialogs** is constructed like this:

```
EXPORT Dialogs.

IMPORT Schemas.

PREDICATE     StrategyInterpreter:
              String                  The name of a schema.
            * String                  The name of the module
                                      containing a strategy S allowing
                                      the instantiation of a schema
                                      having its name equal to the
                                      first argument.
            * Schema.                 An instantiated schema obtained
                                      by the application of the
                                      strategy S to a schema having
                                      its name equal to the first
                                      argument.
```

The local part of the module **Dialogs** is constructed like this:

```
LOCAL Dialogs.

IMPORT Interpreter, ProgramsIO.

StrategyInterpreter(schemaname,stratfile,algo)  <-

ProgramCompile(stratfile,strategy)              &
ConstructGoal(schemaname,goal)                  &
MySucceed(strategy,goal,tsubst)                 &
VariableName(result,''result'',0)               &
BindingInTermSubst(result,resterm,tsubst)       &
TermToSchema(resterm,algo).
```

Remark: The modules imported by **Dialogs** contain services needed to construct the body of the *StrategyInterpreter* predicate.
The *ProgramCompile* predicate produces, in its second argument, the representation of a program when its first argument is the name of a file containing this program. The *ConstructGoal* predicate produces, in its second argument, the representation of the goal

$$CreateSchema(\textbf{schemaname}, schema)\ \&\ Strategy(schema, result)$$

where **schemaname** is obtained via its first argument. The *MySucceed* predicate is a Gödel interpreter using the SLDNF-resolution and the safe leftmost literal computation rule to produce an answer substitution (its third argument) from a Gödel program (its first argument) and a goal (its second argument). The *VariableName* predicate constructs the representation of a variable named "result." The *BindingInTermSubst* predicate extracts from a term substitution (its third argument) the term (its second argment) which is bound to the variable given as first argument. The *TermToSchema* predicate constructs from the representation of a term of type SCHEMA (its first argument) the corresponding schema (its second argument).

The service *Dialogs* is built above the *StrategyInterpreter* predicate presented upper. The local part of the module **Dialogs** is modified as follows:

```
LOCAL Dialogs.

IMPORT Interpreter, ProgramsIO, SchemasCreation, FindStrategies.

StrategyInterpreter(schemaname,stratfile,algo)                <-

            ...

Dialogs()                                                     <-

FetchSchemaName(schemaname)                                    &
schemaname ≠ ''nil''                                           &
FetchStrategyName(stratname)                                   &
stratname ≠ ''nil''                                            &
StrategyInterpreter(schemaname,stratname,algorithm)            &
SchemaToString(algorithm,stralgo)                              &
WriteString(StdOut,stralgo).

Dialogs()                                                     <-

FetchSchemaName(schemaname)                                    &
schemaname ≠ ''nil''                                           &
FetchStrategyName(stratname)                                   &
stratname = ''nil''                                            &
FindStrategyFile(schemaname,stratname)                         &
WriteString(StdOut,''The strategy chosen is '' ++ stratname)   &
StrategyInterpreter(schemaname,stratname,algorithm)            &
SchemaToString(algorithm,stralgo)                              &
WriteString(StdOut,stralgo).
```

```
Dialogs()                                                    <-

FetchSchemaName(schemaname)                                   &
schemaname = ''nil''                                          &
CreateSchema(schemaname,_)                                    &
FindStrategyFile(schemaname,stratname)                        &
WriteString(StdOut,''The schema chosen is '' ++ schemaname)   &
WriteString(StdOut,''The strategy chosen is '' ++ stratname)  &
StrategyInterpreter(schemaname,stratname,algorithm)           &
SchemaToString(algorithm,stralgo)                             &
WriteString(StdOut,stralgo).
```

The predicate *FetchSchemaName* (resp. *FetchStrategyName*) obtains, via a dialog with the user, either the name of the schema (resp. the strategy) the user has chosen or the fact [1] that the user let the synthesizer choose itself a schema (resp. a strategy). The predicate *SchemaToString* is the predicate defined earlier in the section 4 which gives the string representing a schema. The predicate *WriteString* is a predicate writing its second argument on the standard output.

The *FindStrategyFile* predicate belongs to the module **FindStrategies**. This predicate is needed if the user has no preference on the strategy to be used to synthesize a logic algorithm. The *FindStrategyFile* predicate can be viewed as a table linking a strategy to the schema it instantiates.

The export part of the module **FindStrategies** is constructed like this:

```
EXPORT FindStrategies.

IMPORT Strings.

PREDICATE     FindStrategyFile:
              String              The name of a schema.
            * String.             The name of the module
                                  containing a strategy allowing
                                  the instantiation of a schema
                                  having its name equal to the
                                  first argument.
```

The local part of the module **FindStrategies** is constructed like this:

[1] Represented here by a "nil" response of the user.

```
LOCAL FindStrategies.

   FindStrategyFile(schemaname,stratfile)  <-

   (schemaname = ''DC1''                    &   stratfile = ''DC1strat1'')
∨  (schemaname = ''GT''                     &   stratfile = ''GTstrategy'')
∨  (schemaname = ''DC1''                    &   stratfile = ''DC1strat2'')
∨                                          ...
```

Chapter 8

Blocks Improvements

The five blocks composing the synthesizer architecture have been described. We now explain how to add new specification frameworks, new schemas, new methods and new strategies to improve the possibilities offered by the synthesizer **DIALOGS**.

8.1 Specification Block

Due to different reasons, the framework defined previously to specify a predicate is not adequate anymore to synthesize a logic algorithm for a particular predicate p. The following new framework is needed:

- the predicate symbol and the arguments of the predicate for which the user wants a logic algorithm;
- the types of the arguments of this predicate;
- a set of positive examples for which the predicate has to succeed;
- a set of negative examples for which the predicate has to fail;
- a set of available primitive predicates that can be used to synthesize a logic algorithm for the predicate p.

Example: Here is a specification of the predicate *member*/2 in the new framework defined upper:

$$member(E, L)$$

$$E : Term$$
$$L : List(Term)$$

$$member(a, [a])$$
$$member(b, [a, b])$$

$$\neg member(a, [])$$
$$\neg member(b, [a, c, d])$$

$$=, \neq$$

The export part of the module **NewSpecifs** is constructed like this:

```
EXPORT NewSpecifs.

IMPORT Sobjects, Examples, Primitives.

BASE NewSpecif.

PREDICATE     DialogConstructNewSpecif:
              NewSpecif.                  The representation of a
                                          specification extracted
                                          from the standard input via
                                          a dialog with the user.
PREDICATE     ConstructNewSpecif:
              String                      A string.
            * NewSpecif.                  The representation of a
                                          specification extracted
                                          from a file having its root
                                          name equal to the first
                                          argument and its extension
                                          equal to ''nsp.''
```

```
PREDICATE     ComponentsOfNewSpecif:
              NewSpecif                 The representation of a
                                        specification.
          *   String                    The specified predicate.
          *   List(Type)                The list of types of the
                                        arguments of the specified
                                        predicate.
          *   List(Example)             A list of positive examples
                                        for which the specified
                                        predicate has to succeed.
          *   List(Example)             A list of negative examples
                                        for which the specified
                                        predicate has to fail.
          *   List(Primitive).          A list of primitive
                                        predicates that can be used
                                        to synthesize a logic
                                        algorithm for the specified
                                        predicate.
PREDICATE     NewSpecifToString:
              NewSpecif                 The representation of a
                                        specification.
          *   String.                   A string representing this
                                        specification.
```

The imported module **Primitives** contains the abstract data type PRIMITIVE and services to manipulate this abstract data type.
The import part of the module **NewSpecifs** is constructed like this:

```
LOCAL NewSpecifs.

FUNCTION NSpcf:  String * List(Type) * List(Example)
         * List(Example) * List(Primitive) -> NewSpecif.

DialogConstructNewSpecif(spec)                          <-  ...
ConstructNewSpecif(fname,spec)                          <-  ...
ComponentsOfNewSpecif(spec,pred,ltype,pex,nex,lp)       <-  ...
NewSpecifToString(spec,str)                             <-  ...
```

The module **NewSpecifs** is not fundamentally different for the module **Specifs**. It contains services to extract specifications from a file (or from the standard input stream) and services to manipulate specifications. Giving the task of extracting a specification to a

strategy instead of the synthesizer has the advantage that the synthesizer is independent of the framework chosen to specify a predicate. A strategy, being a sequence of methods used to instantiate placeholders in a schema, is naturally dependent on the framework chosen for the specification because each method composing the strategy has specific needs in terms of specification. A method A extracts information from positive examples when a method B extracts information from negative examples only.

Remark: Another advantage of giving to a strategy the task to extract itself a specification is a *specification dialog control*. We implicitly propose a simple interaction, concerning the specification, between the strategy and the user during synthesis. The user gives a specification for a predicate p and, then, the strategy (through methods calls) uses the specification to design a logic algorithm for the predicate p. Other strategies pilot a dialog with the user of the synthesizer to obtain more information about the predicate for which a logic algorithm is under construction. If a need for new negatives examples appears in the middle of the synthesis, the strategy can ask for some to the user. The synthesizer cannot handle beforehand all the variety of possible dialogs with a user. This specification dialog control feature is important with synthesizers using inductive inference as a way to extract useful information from specifications. Inductive inference starts with incomplete and possibly ambiguous information, thus, the need for clarifying information exists.

8.2 Schemas Creation Block

Adding a new schema allowing, for example, the design of logic algorithms in the Global Search synthesis methodology requires a modification of the local part of the module **SchemasCreation**. The export part of this module is not affected by this change and still only contains the declaration of the *CreateSchema*/2 predicate.

The local part of the module **SchemasCreation** is the following:

```
LOCAL SchemasCreation.

CreateSchema(''DC1'',sch)  <-  ...
CreateSchema(''GT'',sch)   <-  ...
CreateSchema(''DC2'',sch)  <-  ...
CreateSchema(''GS'',sch)   <-  ...
```

The name identifying this new schema has to be different from the names of schemas already existing in the module **SchemasCreation**. This requirement is important to avoid problems such as the use of a strategy to instantiate a schema that is not suitable.

Remark: It is necessary to construct, at least, one strategy related to this new Global Search schema. Otherwise, this new schema has no use because there is no indication on how to instantiate its placeholders.

8.3 Methods Block

A method M has been designed to obtain information from negative examples. This method had caused the need for a new specification framework discussed previously. To add the method M to our toolbox, we just have to declare a predicate M in the export part of the module **Methods** and to define it in the local part of the same module. The toolbox containing a greater variety of methods will facilitate the design of future strategies.
Remark: There is no real obligation to add the method M to the module **Methods**. We can create a new module containing this method. A strategy composed of the method M and methods belonging to the module **Methods** must just import these two modules. This feature, allowing to split the set of available methods into several modules, provides a way to structurate the methods according to some criteria.

8.4 Strategies Block

In the previous sections, we have defined a new schema representing the Global Search synthesis methodology, a new method extracting information from negative examples and a new specification framework allowing a user to specify a predicate p with positive examples, negative examples and primitive predicates. We must now construct a strategy instantiating the placeholders of the Global Search schema. This strategy is a Gödel program.
The export part of the module **GSstrategy** representing the strategy instantiating a Global Search schema is constructed like this:

```
EXPORT GSstrategy.

IMPORT NewSpecifs, SchemasCreation, Methods.

PREDICATE     Strategy:
              Schema      The representation of a Global Search schema.
            * Schema.     The schema corresponding to the first
                          argument where all its predicate variables
                          and form variables have been instantiated.
```

To use the new framework of specification defined previously, this strategy has just to import the module **NewSpecifs**.
The local part of the module **GSstrategy** is constructed like this:

```
LOCAL GSstrategy.

Strategy(schema,result)          <-

GetSpecifFileName(fname)          &
ConstructNewSpecif(fname,nspec)   &
MethodA(_,_,_)                    &
...
```

The strategy instantiating a Global Search schema extracts the specification via the *ConstructNewSpecif* service.

Remark: When a strategy is chosen, either by the user or the synthesizer, this strategy must find a file containing the specification of a predicate in the adequate framework. If the user let the synthesizer choose a particular strategy, he is not aware beforehand of the specification framework required by this future strategy. He has to construct as many specifications for his predicate as there are specification frameworks. The *Dialog-ConstructSpecif* service (and other services having the same purpose) allows to avoid this inconvenient.

After the definition of the strategy above allowing the instantiation of a Gloabl Search schema, we must now modify the module **FindStrategies** to make this strategy available to the synthesizer. The body of the *FindStrategyFile* predicate will be the following:

```
LOCAL FindStrategies.

   FindStrategyFile(schemaname,stratfile)  <-

   (schemaname = ''DC1''                    &   stratfile = ''DC1strat1'')
V  (schemaname = ''GT''                     &   stratfile = ''GTstrategy'')
V                                          ...
V  (schemaname = ''GS''                     &   stratfile = ''GSstrategy'').
```

All the modifications and additions presented in this section have not perturbed the functioning of the synthesizer. No redesign of the module **Dialogs** has taken place even if a new specification framework, a new schema, a new method and a new strategy are now available to a user of the synthesizer.

Chapter 9

Conclusion

We have proposed a modular architecture for an inductive schema-guided logic program synthesizer composed of five main modules. We have also described how to add new objects to the main modules [1] without a redesign of the synthesizer.

The synthesizer **DIALOGS** can be incorporated in a logic programs development environment like **FOLON**. **FOLON** is the logic programs development environment supporting the three steps of the methodology explained in [2]. This environment is an integrated set of tools helping a user to:

1. elaborate a specification for a given problem;
2. construct a logic algorithm from the specification in pure logic;
3. derivate a logic program from the specification and the logic algorithm in a specific logic programming language.

Adding the synthesizer **DIALOGS** to the **FOLON** environment allow a user to choose between a manual construction or an automatic construction of a logic algorithm. It will probably require a modification of the specification framework used in the **FOLON** environment. The synthesizer can benefit from the tools already existing in **FOLON** like the type checking, multiplicity checking and directionality checking used to derivate a multidirectional logic program from a logic algorithm.

[1] Excepting the synthesizer module.

Bibliography

[1] A.F. BOWERS. Representing gödel object programs in gödel. Technical Report CSTR-92-31, Department of Computer Science, University of Bristol, 1992.

[2] Y. DEVILLE. *Logic Programming Systematic Program Development.* Addison-Wesley, 1990.

[3] P. FLENER. Logic program schemata: Synthesis and analysis. Technical Report BU-CEIS-9502, Bilkent University, Ankara, 1995.

[4] P. FLENER. *Logic Program Synthesis from Incomplete Information.* Kluwer Academic Publishers, 1995.

[5] C.A. GURR. Specializing the ground representation in the logic programming language gödel. In Y. Deville, editor, *Proceedings of the Third International Workshop on Logic Program Synthesis and Transformation (LOPSTR'93).* Springer-Verlag, 1994.

[6] P. HILL and J. LLOYD. *The Gödel Programming Language.* The MIT Press, 1994.

[7] J.R. HINDLEY and J.P. SELDIN. *Introduction to Combinators and λ-Calculus.* Student Texts. Cambridge University Press, 1986.

[8] J. LLOYD. *Foundations of Logic Programming.* Springer-Verlag, second extended edition, 1987.

[9] G. NADATHUR and D. MILLER. Higher order logic programming. In D. GABBAY, C. HOGGER, and A. ROBINSON, editors, *To be published in The Handbook of Logic in Artificial Intelligence and Logic Programming.* Oxford University Press.

[10] D.R. SMITH. Top-down synthesis of divide-and-conquer algorithms. *Artificial Intelligence*, (27):43–96, 1985.

Appendix

```
EXPORT Sobjects.

% Module providing a number of abstract data types for representing schema
% objects (using the Goedel ground representation) and predicates for
% manipulating these types.
%
% Description of the abstract data types:
%
% Indexed Variables (IVar) are compound variables. The first part is called
% the root variable and the second part the index variable.
% Example: Xj where X is the root variable and j the index variable.
% Remark: the representation of a variable in Goedel is also composed of 2
%         parts. The root which is a string and the index which is an
%         integer. To avoid ambiguities, we will use the following
%         terminology: the index (integer) will be called the mark of the
%         variable.
%
% Vectors are terms composed of 3 parts. The first part is called the root of
% the vector, the second part the lower bound of the vector and the third
% part the upper bound of the vector. The root can either be a variable, an
% indexed variable or a vector. The lower and upper bounds are either
% variables or non-negative integer constants.
% Example: Z [1..n] , W [1..2] [m..p]
%
% Atoms are formulas of the form p(t1,t2,...,tn) where p is called the name
% (predicate variable or predicate symbol) of the atom and t1,t2,...,tn
% the arguments of the atom. The name can either be a variable (representing
% a placeholder) or a constant (representing an instantiated atom).
% The arguments are terms including indexed variables and vectors.
%
% Indexed Atoms (IAtom) are compound atoms of the form p k (t1,t2,...,tn)
% where p is called the name (predicate variable) of the atom, k the index
% and t1,t2,...,tn the arguments of the atom. The name and the index are
% variables. The arguments are terms including indexed variables and vectors.
% Example: Compose k (HY[1..h'],TY[1..t],Y)
%
% Conjunctions (Conj) are formulas composed of 4 parts. the first part is
% called the list of atoms, the second part the index variable, the third
% the lower bound and the fourth the upper bound. The list of atoms can
% either be atoms, indexed atoms, their negation or conjunctions. The lower
% and upper bounds are either variables or non-negative integer constants.
% Example: AND [1=<j=<t] R(TXj,TYj) & V(Z,W) where j is the index variable,
% 1 the lower bound of the conjunction, t the upper bound of the conjunction
% and R(TXj,TYj) , V(Z,W) the list of atoms.
%
% Disjunctions (Disj) are formulas composed of 4 parts. The first part is
% called the list of atoms, the second part the index variable, the third
% the lower bound and the fourth the upper bound. The list of atoms can
% either be atoms, indexed atoms, their negation or conjunctions. The lower
% and upper bounds are either variables or non-negative integer constants.
% Example: OR [1=<j=<t] S(Tj) where j is the index variable,
% 1 the lower bound of the disjunction, t the upper bound of the disjunction
% and S(Tj) the list of atoms.
%
% Abstractions (Abstr) are formulas composed of 2 parts. The first part is
% called the argument of the abstraction and the second is called
% the body of the abstraction. The argument is a variable and the body can
% either be a formula in disjunctive normal form or an abstraction.
% This object is used to instantiate the placeholders of a schema.
```

```
IMPORT ProgramsIO , Numbers .

PREDICATE FetchMyMod :

  Program.              % The program containing the flat representation of the
                        % functions used by the abstract data types.

PREDICATE CreateIVar :

  Program               % The program containing the flat representation of the
                        % functions used by this abstract data type.
* Term                  % The representation of a variable.
* Term                  % The representation of a variable.
* Term.                 % The representation of an indexed variable where the
                        % root is the second argument and the index is the third
                        % argument.

DELAY CreateIVar(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE IsIVar :

  Program               % The program containing the flat representation of the
                        % functions used by this abstract data type.
* Term.                 % The representation of an indexed variable.

DELAY IsIVar(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE VariableOfIVar :

  Term                  % The representation of an indexed variable.
* Term.                 % The root of this indexed variable.

DELAY VariableOfIVar(x,_) UNTIL GROUND(x).

PREDICATE IndexOfIVar :

  Term                  % The representation of an indexed variable.
* Term.                 % The index of this indexed variable.

DELAY IndexOfIVar(x,_) UNTIL GROUND(x).

PREDICATE ReIndexIVar :

  Term                  % The representation of an indexed variable Xj.
* Term                  % The representation of a variable.
* Integer               % A non negative integer.
* Term.                 % The representation of this indexed variable (if
                        % its index j is not equal to the second argument)
                        % or the representation of the root X where its mark
                        % has been modified by the third argument (if the
                        % index j of the indexed variable is equal to the
                        % second argument). The modification of the mark is
                        % as follows: if the mark is equal to 0 then it is
                        % replaced by the third argument. If the mark is
                        % greater than 0 then the new mark is: third argument,
                        % actual mark.

DELAY ReIndexIVar(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE IVarToString :
```

```
  Term                % The representation of an indexed variable.
* String.             % The string representing this indexed variable.

DELAY IVarToString(x,_) UNTIL GROUND(x).

PREDICATE CreateVector :

  Program             % The program containing the flat representation of the
                      % functions used by this abstract data type.
* Term                % The representation of a variable, an indexed variable
                      % or a vector.
* Term                % The representation of a variable or a non-negative
                      % integer constant.
* Term                % The representation of a variable or a non-negative
                      % integer constant.
* Term.               % The representation of a vector where the root is the
                      % second argument, the lower bound is the third argument
                      % and the upper bound is the fourth argument.

DELAY CreateVector(x,y,z,w,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z)
                                  & GROUND(w).

PREDICATE IsVector :

  Program             % The program containing the flat representation of the
                      % functions used by this abstract data type.
* Term.               % The representation of a vector.

DELAY IsVector(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE TermOfVector :

  Term                % The representation of a vector.
* Term.               % The root of this vector.

DELAY TermOfVector(x,_) UNTIL GROUND(x).

PREDICATE LowerBoundOfVector :

  Term                % The representation of a vector.
* Term.               % The lower bound of this vector.

DELAY LowerBoundOfVector(x,_) UNTIL GROUND(x).

PREDICATE UpperBoundOfVector :

  Term                % The representation of a vector.
* Term.               % The upper bound of this vector.

DELAY UpperBoundOfVector(x,_) UNTIL GROUND(x).

PREDICATE VectorInstantiated :

  Program             % The program containing the flat representation of the
                      % functions used by this abstract data type.
* Term.               % The representation of a vector where its lower and
                      % upper bounds are both instantiated (lower and upper
                      % bounds are non-negative integer constants). If the
                      % root is a vector then this vector must also be
```

```
                    % instantiated (and so on recursively).

DELAY VectorInstantiated(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE ExpandVector :

  Program           % The program containing the flat representation of the
                    % functions used by this abstract data type.
* Term              % The representation of an instantiated vector
* List(Term).       % The list [ROOT i] where root is the root of the vector
                    % with lb=<i=<ub where lb and ub are respectively the
                    % lower and upper bounds of this vector. If lb > ub,
                    % the list is empty.

DELAY ExpandVector(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE VectorToString :

  Program           % The program containing the flat representation of the
                    % functions used by this abstract data type.
* Term              % The representation of a vector.
* String.           % The string representing this vector.

DELAY VectorToString(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE CreateAtom :

  Program           % The program containing the flat representation of the
                    % functions used by this abstract data type.
* Term              % The representation of a variable or a constant.
* List(Term)        % A list of representations of terms.
* Formula.          % The representation of an atom where the name of this
                    % atom is the second argument and the arguments of this
                    % atom the third argument.

DELAY CreateAtom(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE IsAtom :

  Program           % The program containing the flat representation of the
                    % functions used by this abstract data type.
* Formula.          % The representation of an atom.

DELAY IsAtom(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE IsInstAtom :

  Program           % The program containing the flat representation of the
                    % functions used by this abstract data type.
* Formula.          % The representation of an instantiated atom (the name
                    % of this atom is a constant and the arguments of this
                    % atom neither contain vectors of variables nor indexed
                    % variables).

DELAY IsInstAtom(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE NameOfAtom :

  Formula           % The representation of an atom.
* Term.            % The name of this atom.
```

```
DELAY NameOfAtom(x,_) UNTIL GROUND(x).

PREDICATE ArgOfAtom :

  Formula              % The representation of an atom.
* List(Term).          % The arguments of this atom.

DELAY ArgOfAtom(x,_) UNTIL GROUND(x).

PREDICATE ReIndexAtom :

  Program              % The program containing the flat representation of the
                       % functions used by this abstract data type.
* Formula             % The representation of an atom.
* Term                % The representation of a variable.
* Integer             % A non negative-integer.
* Formula.            % The representation of this atom where the indexed
                       % variables belonging to the arguments of the atom and
                       % having the same index as the third argument
                       % have been reindexed with the fourth argument.

DELAY ReIndexAtom(x,y,z,w,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z)
                                 & GROUND(w).

PREDICATE ExpandAtom :

  Program              % The program containing the flat representation of the
                       % functions used by this abstract data type.
* Formula             % The representation of an atom.
* Formula.            % The representation of this atom where the instantiated
                       % vectors belonging to the arguments of the atom have
                       % been expanded.

DELAY ExpandAtom(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE AtomToString :

  Program              % The program containing the flat representation of the
                       % functions used by this abstract data type.
* Formula             % The representation of an atom.
* String.             % The string representing this atom.

DELAY AtomToString(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE InstAtomToString :

  Program              % The program containing the flat representation of the
                       % functions used by this abstract data type.
* Formula             % The representation of an instantiated atom.
* String.             % The string representing this instantiated atom.

DELAY InstAtomToString(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE CreateIAtom :

  Program              % The program containing the flat representation of the
                       % functions used by this abstract data type.
* Term                % The representation of a variable.
* Term                % The representation of a variable.
```

```
* List(Term)          % A list of representations of terms.
* Formula.            % The representation of an indexed atom where the name
                      % of this indexed atom is the second argument, the index
                      % is the third and the arguments of this indexed atom
                      % the fourth argument.

DELAY CreateIAtom(x,y,z,w,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z)
                                 & GROUND(w).

PREDICATE IsIAtom :

  Program             % The program containing the flat representation of the
                      % functions used by this abstract data type.
* Formula.            % The representation of an indexed atom.

DELAY IsIAtom(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE NameOfIAtom :

  Formula             % The representation of an indexed atom.
* Term.               % The name of this indexed atom.

DELAY NameOfIAtom(x,_) UNTIL GROUND(x).

PREDICATE IndexOfIAtom :

  Formula             % The representation of an indexed atom.
* Term.               % The index of this indexed atom.

DELAY IndexOfIAtom(x,_) UNTIL GROUND(x).

PREDICATE ArgOfIAtom :

  Formula             % The representation of an indexed atom.
* List(Term).         % The arguments of this indexed atom.

DELAY ArgOfIAtom(x,_) UNTIL GROUND(x).

PREDICATE ReIndexIAtom :

  Program             % The program containing the flat representation of the
                      % functions used by this abstract data type.
* Formula             % The representation of an indexed atom p k (t1,...,tn)
* Term                % The representation of a variable.
* Integer             % A non negative integer.
* Formula.            % The representation of an atom p'(u1,...,un) where
                      % the name p' of this atom is equal to the name p of
                      % the indexed atom with its mark modified by the fourth
                      % argument and where the arguments u1,...,un of this
                      % atom are equal to the arguments t1,...,tn of the
                      % indexed atom reindexed by the third and fourth
                      % argument (if the index k is equal to the third
                      % argument).
                      % Or the representation of an indexed atom
                      % p k (u1,...,un) where its arguments u1,...un are equal
                      % to the arguments t1,...,tn reindexed by the third and
                      % fourth argument (if the index k is not equal to the
                      % third argument).

DELAY ReIndexIAtom(x,y,z,w,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z)
```

```
                                & GROUND(w).

PREDICATE ExpandIAtom :

  Program               % The program containing the flat representation of the
                        % functions used by this abstract data type.
* Formula               % The representation of an indexed atom.
* Formula.              % The representation of this indexed atom where the
                        % instantiated vectors belonging to the arguments of
                        % the indexed atom have been expanded.

DELAY ExpandIAtom(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE IAtomToString :

  Program               % The program containing the flat representation of the
                        % functions used by this abstract data type.
* Formula               % The representation of an indexed atom.
* String.              % The string representing this indexed atom.

DELAY IAtomToString(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE CreateConj :

  Program               % The program containing the flat representation of the
                        % functions used by this abstract data type.
* List(Formula)        % A list containing the representation of atoms, indexed
                        % atoms, their negation and conjunctions.
* Term                 % The representation of a variable.
* Term                 % The representation of a variable or a non-negative
                        % integer constant.
* Term                 % The representation of a variable or a non-negative
                        % integer constant.
* Formula.             % The representation of a conjunction where the list of
                        % atoms is the second argument, the index the third
                        % argument, the lower bound the fourth argument and the
                        % upper bound the fifth argument.

DELAY CreateConj(x,y,z,w,u,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z)
                                & GROUND(w) & GROUND(u).

PREDICATE IsConj :

  Program               % The program containing the flat representation of the
                        % functions used by this abstract data type.
* Formula.             % The representation of a conjunction.

DELAY IsConj(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE ListAtomsOfConj :

  Program               % The program containing the flat representation of the
                        % functions used by this abstract data type.
* Formula               % The representation of a conjunction.
* List(Formula).       % The list of atoms of this conjunction.

DELAY ListAtomsOfConj(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE IndexOfConj :
```

```
  Formula              % The representation of a conjunction.
* Term.               % The index of this conjunction.

DELAY IndexOfConj(x,_) UNTIL GROUND(x).

PREDICATE LowerBoundOfConj :

  Formula              % The representation of a conjunction.
* Term.               % The lower bound of this conjunction.

DELAY LowerBoundOfConj(x,_) UNTIL GROUND(x).

PREDICATE UpperBoundOfConj :

  Formula              % The representation of a conjunction.
* Term.               % The upper bound of this conjunction.

DELAY UpperBoundOfConj(x,_) UNTIL GROUND(x).

PREDICATE ConjInstantiated :

  Formula.             % The representation of a conjunction where its lower
                       % and upper bounds are both instantiated (lower and
                       % upper bounds are integer constants).

DELAY ConjInstantiated(x) UNTIL GROUND(x).

PREDICATE ReIndexConj :

  Program              % The program containing the flat representation of the
                       % functions used by this abstract data type.
* Formula              % The representation of a conjunction.
* Term                 % The representation of a variable.
* Integer              % A non negative integer.
* Formula.             % The representation of this conjunction where the atoms
                       % and indexed atoms belonging to the list of atoms of the
                       % conjunction have been reindexed with the third and
                       % fourth arguments. The third argument must be different
                       % of the index of the conjunction.

DELAY ReIndexConj(x,y,z,w,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z)
                                & GROUND(w).

PREDICATE ExpandConj :

  Program                % The program containing the flat representation of
                         % the functions used by this abstract data type.
* Formula                % The representation of an instantiated conjunction.
                         % /\[lb=<j=<ub] atom1, atom2,..., atomp
* List(List(Formula)).   % The list [list-of-atoms k] where list-of-atoms k is
                         % the list of atoms of the conjunction where all the
                         % atoms have been expanded and reindexed with the
                         % index of the conjunction and k = lb..ub.
                         % The implicit connective of these lists is the
                         % AND-connective.

DELAY ExpandConj(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE ConjToString :
```

```
  Program                % The program containing the flat representation of
                         % the functions used by this abstract data type.
* Formula                % The representation of a conjunction.
* String.                % The string representing this conjunction.

DELAY ConjToString(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE CreateDisj :

  Program                % The program containing the flat representation of
                         % functions used by this abstract data type.
* List(Formula)          % A list containing the representation of atoms,
                         % indexed atoms, their negation and conjunctions.
* Term                   % The representation of a variable.
* Term                   % The representation of a variable or a non-negative
                         % integer constant.
* Term                   % The representation of a variable or a non-negative
                         % integer constant.
* Formula.               % The representation of a disjunction where the list
                         % of atoms is the second argument, the index is the
                         % third argument, the lower bound the fourth and the
                         % upper bound the fifth argument.

DELAY CreateDisj(x,y,z,w,u,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z)
                                  & GROUND(w) & GROUND(u).

PREDICATE IsDisj :

  Program                % The program containing the flat representation of
                         % the functions used by this abstract data type.
* Formula.               % The representation of a disjunction.

DELAY IsDisj(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE ListAtomsOfDisj :

  Program                % The program containing the flat representation of
                         % the functions used by this abstract data type.
* Formula                % The representation of a disjunction.
* List(Formula).         % The list of atoms of this disjunction.

DELAY ListAtomsOfDisj(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE IndexOfDisj :

  Formula                % The representation of a disjunction.
* Term.                  % The index of this disjunction.

DELAY IndexOfDisj(x,_) UNTIL GROUND(x).

PREDICATE LowerBoundOfDisj :

  Formula                % The representation of a disjunction.
* Term.                  % The lower bound of this disjunction.

DELAY LowerBoundOfDisj(x,_) UNTIL GROUND(x).

PREDICATE UpperBoundOfDisj :

  Formula                % The representation of a disjunction.
```

```
* Term.                  % The upper bound of this disjunction.

DELAY UpperBoundOfDisj(x,_) UNTIL GROUND(x).

PREDICATE DisjInstantiated :

  Formula.               % The representation of a disjunction where the lower
                         % and upper bounds have been instantiated.

DELAY DisjInstantiated(x) UNTIL GROUND(x).

PREDICATE ExpandDisj :

  Program                % The program containing the flat representation of
                         % the functions used by this abstract data type.
* Formula                % The representation of an instantiated disjunction.
* List(List(Formula)).   % The list [list-of-atoms k] where list-of-atoms k is
                         % the list of atoms of the disjunction where all the
                         % atoms have been expanded and reindexed with the
                         % index of the disjunction and k = lb..ub.
                         % The implicit connective of these lists is the
                         % OR-connective.

DELAY ExpandDisj(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE DisjToString :

  Program                % The program containing the flat representation of
                         % the functions used by this abstract data type.
* Formula                % The representation of a disjunction.
* String.                % The string representing this disjunction.

DELAY DisjToString(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE CreateAbstraction :

  Program                % The program containing the flat representation of
                         % functions used by this abstract data type.
* Term                   % The representation of a variable.
* Formula                % The representation of a formula in disjunctive
                         % normal form or the representation of an abstraction.
* Formula.               % The representation of an abstraction where the
                         % argument of this abstraction is the second argument
                         % and the body of this abstraction the third argument.

DELAY CreateAbstraction(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE IsAbstraction :

  Program                % The program containing the flat representation of
                         % the functions used by this abstract data type.
* Formula.               % The representation of an abstraction.

DELAY IsAbstraction(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE VariableOfAbstraction :

  Formula                % The representation of an abstraction.
* Term.                  % The argument of this abstraction.
```

```
DELAY VariableOfAbstraction(x,_) UNTIL GROUND(x).

PREDICATE FormulaOfAbstraction :

  Program                 % The program containing the flat representation of
                          % the functions used by this abstract data type.
* Formula                 % The representation of an abstraction.
* Formula.                % The body of this abstraction.

DELAY FormulaOfAbstraction(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE CreateNilConstant :

  Program                 % The program containing the flat representation of
                          % the name of the constant nil.
* Term.                   % The representation of the constant nil.

DELAY CreateNilConstant(x,_) UNTIL GROUND(x).

PREDICATE CreateCons :

  Program                 % The program containing the flat representation of
                          % the name of the function Cons.
* Term                    % The representation of a term.
* Term                    % The representation of a term.
* Term.                   % The representation of the list where the head of
                          % the list is the second argument and the tail of the
                          % list is the third argument.

DELAY CreateCons(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE CreateTrue :

  Program                 % The program containing the flat representation of
                          % the name of the proposition true.
* Formula.                % The representation of the proposition true.

DELAY CreateTrue(x,_) UNTIL GROUND(x).

PREDICATE CreateFalse :

  Program                 % The program containing the flat representation of
                          % the name of the proposition false.
* Formula.                % The representation of the proposition false.

DELAY CreateFalse(x,_) UNTIL GROUND(x).

PREDICATE CreateEqualPredicate :

  Program                 % The program containing the flat representation of
                          % the name of the predicate equal.
* Term                    % The representation of a term.
* Term                    % The representation of a term.
* Formula.                % The representation of the predicate equal where
                          % the arguments of this predicate are the second and
                          % the third argument.

DELAY CreateEqualPredicate(x,y,z,_) UNTIL GROUND(x) & GROUND(y) & GROUND(z).

PREDICATE ConstantToInteger :
```

```
  Program               % The program containing the flat representation
                        % of the name of integers.
* Term                  % The representation of an integer constant.
* Integer.              % the integer corresponding to this constant.

DELAY ConstantToInteger(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE IntegerToConstant :

  Program               % The program containing the flat representation
                        % of the name of integers.
* Integer               % An integer.
* Term.                 % The representation of the corresponding integer
                        % constant.

DELAY IntegerToConstant(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE ConstantToString :

  Program               % The program containing the flat representation
                        % of the name of strings.
* Term                  % The representation of a string constant.
* String.               % The string corresponding to this constant.

DELAY ConstantToString(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE StringToConstant :

  Program               % The program containing the flat representation
                        % of the name of strings.
* String                % A string.
* Term.                 % The representation of the corresponding string
                        % constant.

DELAY StringToConstant(x,y,_) UNTIL GROUND(x) & GROUND(y).
```

```
EXPORT Schemas.

% Module providing the abstract data type Schema (using the Goedel
% ground representation) and predicates for manipulating this type.

% Definitions:
%
% An instantiated schema is a schema where the objects VECTOR, CONJUNCTION
% and DISJUNCTION (available in the module Sobjects) have been expanded
% and where all the placeholders have been instantiated.
%
% Given a term T, VAR(T) is defined inductively as follows:
%
%   VAR(T) = {}  if T is a constant;
%   VAR(T) = {T} if T is a variable;
%   VAR(T) = UNION for i=1..n VAR(Ti) where T is a compound term
%                                                  f(T1,T2,...,Tn).
%
% Given a substitution S = {X1/T1 , X2/T2 , ... , Xn/Tn}
% RANGE(S) is the UNION for i=1..n VAR(Ti)
%
% Given a substitution S = {X1/T1 , X2/T2 , ... , Xn/Tn},
% S is idempotent
% IFF the intersection between {X1 , X2 , ... , Xn} and RANGE(S) is empty.


IMPORT Sobjects.

BASE Schema,          % Type of a term representing a schema.
     SchemaSubst.     % Type of a term representing a schema substitution.

PREDICATE FormulaSchema :

  Formula             % The representation of a formula of the form
                      % HEAD <-> BODY where HEAD is an atom
                      % and BODY a well-formed formula in disjunctive normal
                      % form.
* Schema.             % The representation of the schema corresponding to
                      % this formula.

PREDICATE FindAtomInSchema :

  Schema              % The representation of a schema.
* Term                % The predicate variable or predicate symbol of an atom.
                      % It can either be a variable or a constant respectively.
* List(Formula).      % The list of the atoms belonging to the schema
                      % and having their predicate variable or predicate symbol
                      % equal to the second argument.

DELAY FindAtomInSchema(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE InstantiatedSchema :

  Schema.             % The representation of an instantiated schema.

DELAY InstantiatedSchema(x) UNTIL GROUND(x).
```

```
PREDICATE SchemaToGoedel :

  Schema              % The representation of an instantiated schema.
* List(String)        % The list of the types representing the type
                      % declaration of the instantiated schema. The order in
                      % this list of types must be the same as the order of
                      % the arguments of the head part of the schema.
* List(String)        % The list of the names of the modules that will have to
                      % be in the import part of the goedel representation of
                      % the instantiated schema.
* String.             % The root name of the files that will contain the goedel
                      % representation of the instantiated schema. The
                      % extension of the files will be EXP for the file
                      % containing the export part and LOC for the file
                      % containing the local part.

% This predicate has a side-effect which consists in creating the files that
% will contain the goedel representation of the instantiated schema.

DELAY SchemaToGoedel(x,y,z,w) UNTIL GROUND(x) & GROUND(y) & GROUND(z)
                                  & GROUND(w).

PREDICATE SchemaToProlog :

  Schema              % The representation of an instantiated schema.
* String.             % The root name of the file that will contain the prolog
                      % representation of the instantiated schema. The
                      % extension of this filename will be PRO.

% This predicate has a side-effect which consists in creating the file that
% will contain the prolog representation of the instantiated schema.

DELAY SchemaToProlog(x,y) UNTIL GROUND(x) & GROUND(y).

PREDICATE SchemaToString :

  Schema              % The representation of a schema.
* String.             % The string representing this schema (ASCII lower-128).

DELAY SchemaToString(x,_) UNTIL GROUND(x).

PREDICATE InstSchemaToString :

  Schema              % The representation of a schema.
* String.             % The string representing the schema where only appear
                      % the placeholders which have been instantiated.

DELAY InstSchemaToString(x,_) UNTIL GROUND(x).

PREDICATE EmptySchemaSubst :

  SchemaSubst.        % Representation of the empty schema substitution.
```

```
PREDICATE ApplySubstToSchema :

  Schema              % Representation of a schema.
* SchemaSubst         % Representation of an idempotent schema substitution.
* Schema.             % Representation of the schema obtained by applying
                      % this substitution to this schema.

DELAY ApplySubstToSchema(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE ComposeSchemaSubsts :

  SchemaSubst         % Representation of a schema substitution where the
                      % intersection between {X1,X2,...,Xn} belonging to the
                      % first argument and RANGE(secon argument) must be empty.
* SchemaSubst         % Representation of a schema substitution.
* SchemaSubst.        % Representation of the substitution obtained by
                      % composing these two schema substitutions.
                      % third argument = second argument  o  first argument.

DELAY ComposeSchemaSubsts(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE BindingSVarToSchemaSubst :

  Term                % Representation of a schema variable.
* Term                % Representation of a non-negative integer constant.
* SchemaSubst.        % Representation of the schema substitution containing
                      % just the binding in which this schema variable is
                      % bound to this constant.

DELAY BindingSVarToSchemaSubst(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE BindingVarToSchemaSubst :

  Term                % Representation of a variable.
* Term                % Representation of a term.
* SchemaSubst.        % Representation of the schema substitution containing
                      % just the binding in which this variable is bound
                      % to this term.

DELAY BindingVarToSchemaSubst(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE BindingLTermToSchemaSubst :

  Term                % Representation of the predicate variable of an atom.
                      % It has to be a variable.
* Formula             % Representation of a well-formed formula in disjunctive
                      % normal form or representation of an abstraction
                      % (this object is available in the module Sobjects).
* SchemaSubst.        % Representation of the schema substitution containing
                      % just the binding in which this predicate variable is
                      % bound to the second argument.

DELAY BindingLTermToSchemaSubst(x,y,_) UNTIL GROUND(x) & GROUND(y).

PREDICATE IdempotentSchemaSubst :

  SchemaSubst.        % Representation of an idempotent schema substitution.

DELAY IdempotentSchemaSubst(x) UNTIL GROUND(x).
```